ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

BALLOON BREAKERS

Jesús González Rodríguez

13 de octubre de 2011

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

BALLOON BREAKERS

- Departamento: Lenguajes y sistemas informáticos
- Directores del proyecto: Manuel Palomo Duarte y María del Carmen de Castro Cabrera
- Autor del proyecto: Jesús González Rodríguez

Cádiz, 13 de octubre de 2011

Fdo: Jesús González Rodríguez

Agradecimientos

Me gustaría agradecer a mi familia y amigos todo su apoyo durante el desarrollo del proyecto y la carrera.

También me gustaría agradecer a Manuel Palomo su paciencia y apoyo durante todo el desarrollo del proyecto. Gracias a María del Carmen de Castro por sus consejos con la documentación.

Mi agradecimiento también a la comunidad del software libre y de obras copyleft en general sin la cual este proyecto habría sido imposible de realizar para mí.

Gracias también a Kathi y su familia por su apoyo durante los meses que dediqué al proyecto en su país.

Licencia

Notación y formato

Al escribir este texto se ha empleado un conjunto de convenios tipográficos para facilitar la lectura y comprensión del documento:

Cuando nos refiramos a un programa en concreto, utilizaremos la notación:
Programa.

Cuando nos refiramos a un comando, o función de un lenguaje, usaremos la notación:
`comando`.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. El videojuego Balloon Breakers

Los videojuegos se han convertido en un elemento muy importante en nuestra sociedad y forman parte de la vida cotidiana de millones de usuarios. La industria de los videojuegos ha llegado incluso a equipararse a la de del cine y el desarrollo de algunos videojuegos llega incluso a superar en coste a algunas superproducciones de Hollywood.

El objetivo de este PFC es crear un videojuego en 3D basado en el clásico "Pang!".

El juego se divide en niveles donde el jugador debe controlar a un personaje sobre una superficie rectangular donde aparecen burbujas (o bolas), cubos y animales (pájaros carpintero y cangrejos ermitaños).

El jugador pasa de nivel cuando todas las bolas hayan sido destruidas, bien por el mismo jugador o por los animales. El jugador pierde una vida cuando sea alcanzado por una bola o un animal o bien cuando el tiempo del nivel donde este jugando se haya agotado.

En total, el juego esta compuesto por 15 niveles diferentes que muestran 5 localizaciones distintas en 3 periodos del día: mañana, tarde y noche. Cada nivel tiene una velocidad, bolas, animales y cubos diferente.

Se puede observar a continuación el logotipo del juego:

Figura 1.1: Logotipo de *Balloon Breakers*

1.2. Estructura del documento

Este documento se divide en nueve capítulos, el primero de ellos es en el que nos encontramos.

En el segundo capítulo se describen conceptos básicos explicando qué es un videojuego así como las diferentes tecnologías utilizadas durante el desarrollo del proyecto.

En el tercer capítulo se explican las tareas realizadas y el tiempo que se ha dedicado a cada una de ellas. Se incluye un diagrama de Gantt donde se detalla el tiempo dedicado a cada tarea.

En el cuarto capítulo se explica la fase de análisis que se ha realizado en el proyecto. Se incluyen diagramas entidad-relación, más conocidos como ERs.

En el quinto capítulo se expone la fase de diseño del videojuego, es decir, se define el sistema con todo detalle.

En el sexto capítulo se habla sobre la implementación del videojuego, las técnicas y herramientas usadas, así como las diferentes estrategias que se han seguido para realizarlo.

En el séptimo capítulo se muestran las diferentes pruebas a las que ha sido sometido el videojuego para verificar que se cumplen los requisitos del mismo.

En el octavo capítulo expongo las conclusiones a las que he llegado después de la finalización del proyecto, lo que he aprendido, posibles extensiones del proyecto, etc.

En el noveno y último capítulo se incluyen distintos apéndices tales como el manual de usuario del juego. Al ser un producto de Software Libre se incluye el texto de la licencia en la que se basa la documentación del proyecto.

Capítulo 2

Conceptos básicos

2.1. ¿Qué es un videojuego?

Según Wikipedia [2] un videojuego o juego de vídeo es un software creado para el entretenimiento en general y basado en la interacción entre una o varias personas y un aparato electrónico que ejecuta dicho videojuego.

Así pues un videojuego es considerado un programa software cuyo fin directo es el entretenimiento.

Los videojuegos se han hecho cada vez más populares. Desde finales de los años 1970 se han lanzado diferentes videoconsolas y actualmente abarca un amplio mercado. Compañías como Nintendo, Sony, Microsoft... han lanzado las más exitosas videoconsolas y multitud de compañías desarrollan juegos para sus sistemas.

2.1.1. Tipos de videojuegos

Existen diferentes clasificaciones de videojuegos dependiendo de su temática, número de jugadores, etcétera.

Una clasificación actualizada se puede encontrar en la Wikipedia:

1. **Aventura:** Normalmente el jugador encarna el rol de un personaje que debe resolver puzles, derrotar enemigos, etcétera para avanzar en la historia del juego. Las diferentes partes de “The Legend Of Zelda” son un buen ejemplo de videojuegos de aventura.

2. **Disparos:** El jugador controla un personaje, nave, tanque... que debe disparar a los enemigos que aparecen en cada nivel. “Quake”, “Doom” o clásicos como “Galaga” y “Space Invaders” son ejemplos de juegos de disparos.

3. **Educativos:** Son juegos en los que el jugador aprende sobre una o varias temáticas. “Brain Training” así como diversos juegos de idiomas son videojuegos educativos.

4. **Estrategia:** El jugador debe hacer uso de recursos y personajes para llegar a un cierto fin como puede ser destruir los recursos y personajes del enemigo, crear una civilización, etcétera. “Age Of Empires” de Microsoft es un ejemplo del clásico videojuego de estrategia.

5. **Lucha:** El jugador controla un luchador que se enfrenta cuerpo a cuerpo contra otro. La saga "Street Fighter", "Tekken" y "Mortal Kombat" representan buenos ejemplos de videojuegos de lucha.

6. **Survival horror:** Son juegos del género de terror. Juegos como "Resident Evil" y "Alone In The Dark" son videojuegos de este género.

7. **Plataformas:** En estos juegos hay que saltar, subir y bajar plataformas avanzando de nivel cuando se llega a un cierto punto. El clásico "Super Mario Bros." o "Sonic The Hedgehog" son videojuegos de plataformas.

8. **Rol:** Similares a los juegos de aventura pero con más estrategia. El jugador sube de nivel a su o sus personajes, debe utilizar objetos, hechizos, etcétera para avanzar en la historia. La saga "Final Fantasy" es una famosa representante de este género.

9. **Musicales:** El jugador debe hacer uso de sus habilidades musicales como cantar, tocar algún tipo de instrumento, etcétera. "Guitar Hero" o "Sing Star" son videojuegos musicales muy populares.

10. **Party games:** Son juegos para ser jugado en compañía. Normalmente se componen de minijuegos y se realizan competiciones entre los jugadores. Un buen ejemplo de este tipo de juegos es "Mario Party".

11. **Simulación:** En estos juegos se simula una experiencia como puede ser conducir un vehículo (coche, avión...). "Gran Turismo" es un videojuego de simulación de conducción. Aunque el género es muy amplio y abarca otros juegos como "Los Sims", "Dogz", "Sim City"...

12. **Deportivo:** Son juegos que simulan un deporte como puede ser fútbol, baloncesto, juegos olímpicos, juegos de invierno, etcétera. Algunos ejemplos son "Pro Evolution Soccer", "Virtua Tennis" o "NBA Live"

13. **Carreras:** Son juegos, generalmente de coches, en los que se juegan carreras entre los jugadores. "Mario Kart" o el futurista "Wipeout" son juegos de carreras.

2.1.2. El Pang! original

El videojuego *Pang!*, cuyo nombre original es "Buster Bros." fue lanzado al mercado de las recreativas en el año 1989 por la compañía *Capcom*. Tuvo una gran aceptación y ha sido versionado para distintas plataformas como "Super NES", "PlayStation", "Game Boy", ordenadores como "Amiga", "Spectrum", "Commodore 64" y "MS-DOS" e incluso versiones móviles como la de "iPhone" entre otras.

A continuación se pueden observar algunas capturas de pantalla de dichos juegos:

Figura 2.1: Captura de pantalla de *Pang!* en su versión para recreativas

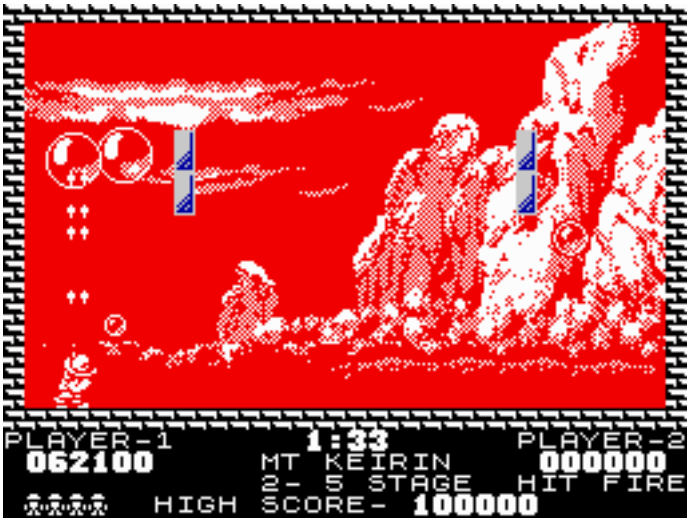

Figura 2.2: Captura de pantalla de *Pang!* en su versión para Spectrum

Es un videojuego mezcla de plataformas y disparos en 2D, de uno o dos jugadores en el cual estos deben destruir las bolas que botan por la pantalla, esquivando animales, para avanzar de nivel.

Cada jugador empieza el juego armado con un arpón que al disparar sube por la pantalla hasta tocar algún elemento.

Consta de 50 niveles recorriendo 17 famosas localizaciones del mundo entero desde el Mt. Fuji en Japón hasta los Rapa Nui de la Isla de Pascua.

Buster Bros. está posiblemente inspirado en un juego 6 años más antiguo llamado *Cannon Ball* para *MSX* y *ZX Spectrum*.

A continuación se puede apreciar una captura de pantalla de dicho juego:

Figura 2.3: Captura de pantalla de *Cannon Ball* en su versión para *MSX* [9]

El juego ha tenido numerosas secuelas oficiales:

- **Super Buster Bros** (también conocido como *Super Pang* 1990)
- **Buster Buddies** (también conocido como *Pang 3* 1995)
- **Buster Bros. Collection** (PlayStation, 1997).
- **Mighty! Pang** (2000).
- **Pang: Magical Michael** (Nintendo DS, 2010)

Sin embargo, hasta la fecha no ha habido ninguna secuela oficial en 3D. Aunque existen algunos remakes en 3D no oficiales, la mayoría no son Software Libre o se alejan bastante de la mecánica y ambientación original habiendo algunos en primera persona, otros con personajes como alienígenas o incluso clones de Star Wars y algunos otros con una estética más "realista" y con escenarios enormes que no conseguían un grado de jugabilidad aceptable. Es por esto por lo que, aparte de mi interés por los videojuegos y en concreto por el *Pang!* original, decidí desarrollar este videojuego.

2.2. Tecnologías implicadas

En esta sección comento las herramientas que he utilizado para el desarrollo del proyecto:

- **C++:** Lenguaje de programación orientado a objetos. Todo el código fuente de *Balloon Breakers* ha sido escrito en C++.

- **Ogre 3D:** Motor de renderizado 3D orientado a escenas, escrito en el lenguaje de programación C++. He usado Ogre 3D para todo el apartado gráfico del juego (menús, modelos 3D, escenarios, texturas, luces, sombras...).
- **OIS:** Conjunto de bibliotecas en C++ para la gestión de la entrada. Ogre 3D lo integra para la utilización del ratón y teclado entre otros.
- **SDL y SDL mixer:** SDL es un conjunto de bibliotecas muy utilizado en el desarrollo de videojuegos en 2D. Es libre y existen multitud de videojuegos libres que lo utilizan. SDL mixer es la parte de SDL encargada de gestionar la música y el sonido. Utilicé SDL mixer para la gestión de la música y del sonido del juego.
- **TinyXML:** Conjunto de bibliotecas escrito en C++ para la gestión de archivos XML. Utilicé TinyXML para la carga de niveles del juego.
- **Blender:** Blender es un programa informático multiplataforma, dedicado especialmente al modelado, animación y creación de gráficos tridimensionales. He usado Blender para la creación de los modelos 3D que posteriormente he exportado al formato de OGRE 3D.
- **GIMP:** Programa de edición de imágenes digitales en forma de mapa de bits, tanto dibujos como fotografías. Es un programa libre y gratuito y lo he usado para la creación de los elementos 3D del juego tales como texturas y fondos.
- **Audacity:** Aplicación informática multiplataforma libre, que se puede usara para grabación y edición de audio, distribuido bajo la licencia GPL. Lo he usado para la edición de los sonidos y la música del juego.
- **DIA:** Software libre para la creación de diagramas de diferentes tipos. En este proyecto he usado DIA para la creación de los diferentes diagramas en el análisis y el diseño exceptuando el diagrama de Gantt de la planificación.
- **Planner:** Herramienta libre multiplataforma para la gestión y planificación de proyectos. He usado Planner para el documentar en forma de diagramas de Gantt el tiempo dedicado a cada tarea.
- **LaTeX:** Sistema de composición de textos, orientado especialmente a la creación de libros, documentos científicos y técnicos que contengan fórmulas matemáticas. He usado LaTeX para la creación de esta memoria.

Capítulo 3

Planificación

3.1. Incrementos

Al desarrollar *Balloon Breakers* he optado por utilizar el modelo incremental[4] dada la magnitud (para mí) del proyecto. He querido siempre tener una idea de cómo el proyecto ha ido avanzando y una vez obtenido un resultado claro, analizarlo e incrementarlo.

Se puede dividir el proyecto en los tres incrementos que explicaré a continuación. Hay que tener en cuenta que paralicé el proyecto entre el primer y el segundo incremento durante algunos meses.

3.1.1. Incremento 1: Personaje y escena básica

En este primer incremento realicé un primer ejecutable donde se muestra el personaje animado sobre una superficie rectangular donde se puede mover al personaje.

Este primer incremento me ha servido para estimar la dificultad del proyecto y el tiempo necesario para realizar el juego al completo.

He trabajado unas 5 horas aproximadamente de lunes a viernes durante 5 meses durante las cuales he aprendido a utilizar *Blender* y *Ogre 3D*. Para el aprendizaje de *Blender*me he guiado por el libro Blender. Curso de Iniciación [5]. Para *Gimp* me guié por el libro Gimp: The Official Handbook [10]. Para el aprendizaje de *Ogre 3D* he seguido los tutoriales, manual y foros de la página de *Ogre 3D*[13] junto al libro *Pro OGRE 3D Programming*[8], refrescando mis conocimientos de *C++* gracias al libro *Fundamentos de C++*[6].

Decidí a utilizar *C++* porque conocía dicho lenguaje de cursos previos en la universidad y por su uso extendido en el desarrollo de videojuegos.

La idea de seleccionar *Ogre 3D* como motor gráfico no fue tan fácil y estuve en principio tanteando otras posibilidades como *Panda3D*[1]. Pese a las primeras dificultades para instalar *Ogre 3D* y hacerlo funcionar me decanté por este motor gráfico debido a su buen funcionamiento y su amplia comunidad.

[1]Panda 3D es un motor gráfico y como tal, proporciona un conjunto de funciones, clases y estructuras de datos para el desarrollo de videojuegos y renderizado en tres dimensiones.

Figura 3.1: Logo de *Ogre 3D*

Blender es el único programa de modelado 3D gratuito con el que estaba familiarizado y, si bien su interfaz es algo complicada al principio, aprendí a utilizarlo con bastante rapidez para hacer el primer personaje con texturas y animaciones y exportarlo al formato de *Ogre 3D*.

Figura 3.2: Logo de *Blender*

Así pues, creé el personaje principal del juego con sus texturas y animaciones y lo exporté al formato de *Ogre 3D*. Siguiendo los tutoriales de *Ogre 3D* creé un pequeño programa de prueba en el que se cargaba al personaje y se le podía mover mediante el teclado.

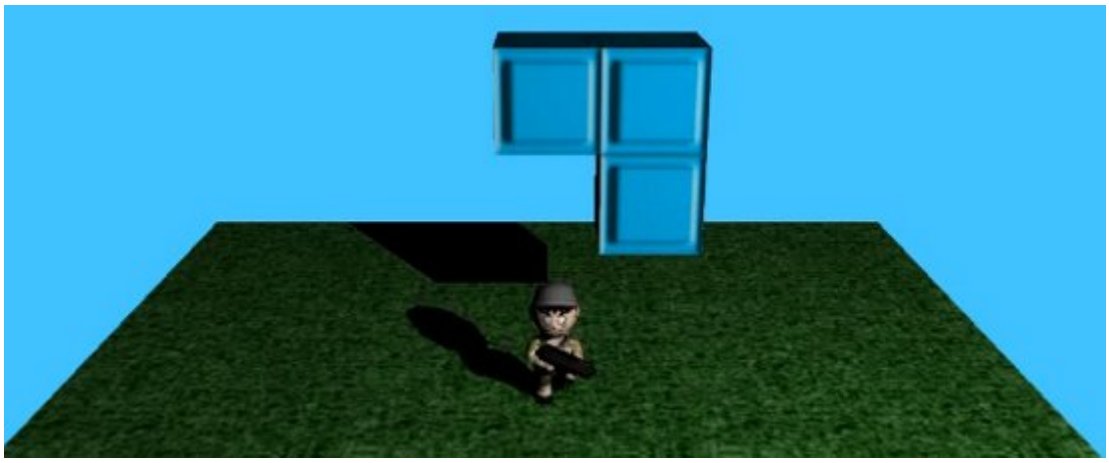

Figura 3.3: Resultado del primer incremento

En definitiva, aprendí a utilizar los elementos básicos de *Ogre 3D* como son la clases `Root`, `SceneManager`, `Entity`, `SceneNode`, `Camera`, `Light` y `Material`.

3.1.2. Incremento 2: Menús, lógica y física del juego

Una vez obtenidos los conocimientos básicos que necesitaba continué con el desarrollo del juego. Diseñé para el segundo incremento el arpón que el personaje dispara así como las bolas y animales que

componen el juego.

Para los menús y mensajes en 2D cómo el número del nivel, vidas, puntuación, etcétera, aprendí a crear las capas u `Overlays` que *Ogre 3D* provee. Tenía la opción de utilizar *myGUI* o *CeGUI* para los menús pero al final decidí que era menos costoso crear mi propio sistema de gestión de menús ya que estos son muy simples en *Balloon Breakers*

Aprendí a cambiar de una escena a otra y fui modularizando todo lo que tenía del primer incremento para hacer el código más legible y entendible.

En cuanto a la lógica del juego creé algunas nuevas clases como la clase `Partida` la cual contiene la información necesaria de una partida del juego como el número de vidas, el nivel y puntuación actual, etcétera. También implementé la carga de niveles a partir de ficheros XML mediante las bibliotecas de *TinyXML*.

Para la física del juego estuve pensando en utilizar *Bullet*[2] y estudié durante algún tiempo cómo utilizarlo así como sus cualidades y, sobre todo, cómo integrarlo con *Ogre 3D*. Al final decidí que, debido a que la física de *Balloon Breakers* no debía ser completamente realista[3] sería menos costoso crear mis propias funciones para la simulación de física (colisiones incluidas). Esto al principio fue bastante simple ya que las colisiones entre esferas (bolas) se resuelven de una manera muy sencilla pero esta decisión me dio algunos problemas más complicados de resolver en el tercer incremento cuando incluí cubos en la escena.

Figura 3.4: Logo de *Bullet*

Finalmente, logré realizar una primera versión del juego en la que aparece un menú, los créditos y una primera versión de una fase sin entorno (aunque la lógica del juego podría cargar un número indeterminado de fases).

[2]Bullet es un conjunto de bibliotecas libre para la simulación de física muy utilizado en videojuegos y otros programas de simulación.

[3]En *Balloon Breakers* las bolas nunca dejan de botar y rebotan con una velocidad diferente dependiendo de la superficie que toquen.

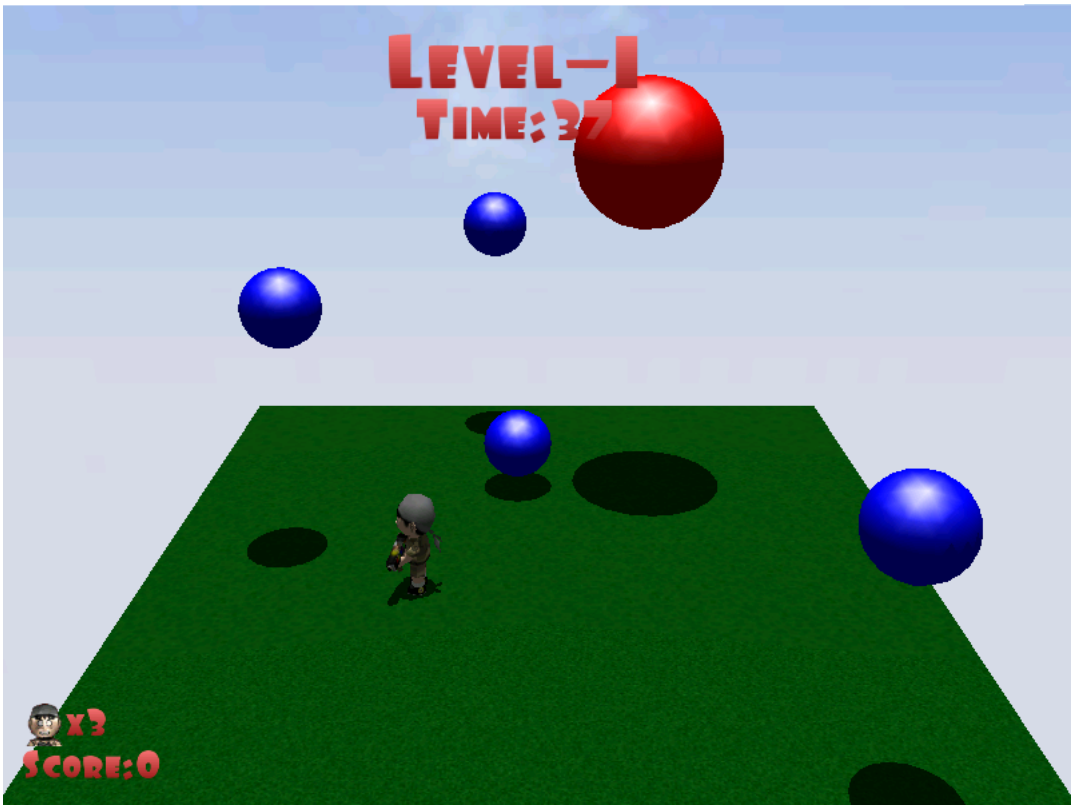

Figura 3.5: Resultado del segundo incremento

3.1.3. Incremento 3: Música, sonido, partículas, cubos y animales

Para finalizar quise añadir escenarios diferentes (un total de cinco), así cómo efectos de partículas y, por supuesto el sonido y la música. El director del proyecto me aconsejó añadir más elementos como plataformas y animales de manera similar al *Pang!* original.

Así pues diseñé los elementos necesarios y añadí las funciones necesarias para controlar las colisiones entre bolas y cubos así como las colisiones entre animales y cubos y animales y bolas.

Para el sonido decidí utilizar *SDL mixer* ya que había oído hablar de él y me pareció una manera fácil, rápida y con buenos resultados. Reforcé mis conocimientos con el libro Focus On SDL [11]. Utilicé voces y sonidos que edité con *Audacity* guiándome por el libro The Book of Audacity [12].

Figura 3.6: Logo de *SDL*

Busqué los sonidos en la página *FreeSound.org* [3] y la música en *Jamendo*. Al final decidí que el tema *Jump!* [7] del compositor *No hair on head* sería el tema principal del juego. Para las fases quería utilizar algo alegre con voces y encontré también en *Jamendo* al grupo japonés *Atomu* [1]. Las voces fueron grabadas por mi compañero y amigo Joaquín Jurado. Una vez tuve todos los archivos los edité cuando fue necesario con el programa *Audacity*.

El resultado de este tercer incremento es la versión estable actual de *Balloon Breakers* que presento en este documento.

Figura 3.7: Resultado del tercer incremento

3.2. Diagrama de Gantt

A continuación se presentan tres tablas y tres diagramas de Gantt con las diferentes tareas de cada incremento del proyecto. Posteriormente se incluye un diagrama de Gantt con el contenido de todo el proyecto incluyendo esta memoria.

WBS	Nombre	Trabajo
1	Primer incremento	83d
1.1	Formación	50d
1.1.1	Aprendizaje de Ogre 3D	25d
1.1.2	Aprendizaje de OIS	2d
1.1.3	Aprendizaje de Blender	20d
1.1.4	Aprendizaje de Gimp	3d
1.2	Análisis del sistema	3d
1.3	Diseño del sistema	5d
1.4	Diseño gráfico: Personaje	15d
1.5	Programación en C++	5d
1.6	Pruebas	5d

Tabla 3.1: Tareas del primer incremento

Figura 3.8: Diagrama de Gantt - Primer incremento

WBS	Nombre	Trabajo
2	Segundo incremento	65d
2.1	Formación	11d
2.1.1	Aprendizaje de Ogre 3D	10d
2.1.2	Aprendizaje de TinyXML	1d
2.2	Análisis del sistema	5d
2.3	Diseño del sistema	5d
2.4	Implementación	39d
2.4.1	Diseño gráfico	17d
2.4.1.1	Bolas	1d
2.4.1.2	Arpón	1d
2.4.1.3	Logo	1d
2.4.1.4	Playa	4d
2.4.1.5	Montaña	3d
2.4.1.6	Desierto	3d
2.4.1.7	Montaña nevada	1d
2.4.1.8	Ciudad	3d
2.4.2	Programación	22d
2.4.2.1	Makefile	1d
2.4.2.2	Overlays de Ogre	2d
2.4.2.3	Programación en C++	19d
2.4.2.3.1	Clase Juego	1d
2.4.2.3.2	Clase MenuPrincipal	1d
2.4.2.3.3	Clase OyenteMenuPrincipal	1d
2.4.2.3.4	Clase Creditos	1d
2.4.2.3.5	Clase OyenteCreditos	1d
2.4.2.3.6	Clase Fase	5d
2.4.2.3.7	Clase OyenteFase	5d
2.4.2.3.8	Clase Personaje	1d
2.4.2.3.9	Clase Bola	1d
2.4.2.3.10	Clase Arpon	1d
2.4.2.3.11	Clase Partida	1d
2.5	Pruebas	5d

Tabla 3.2: Tareas del segundo incremento

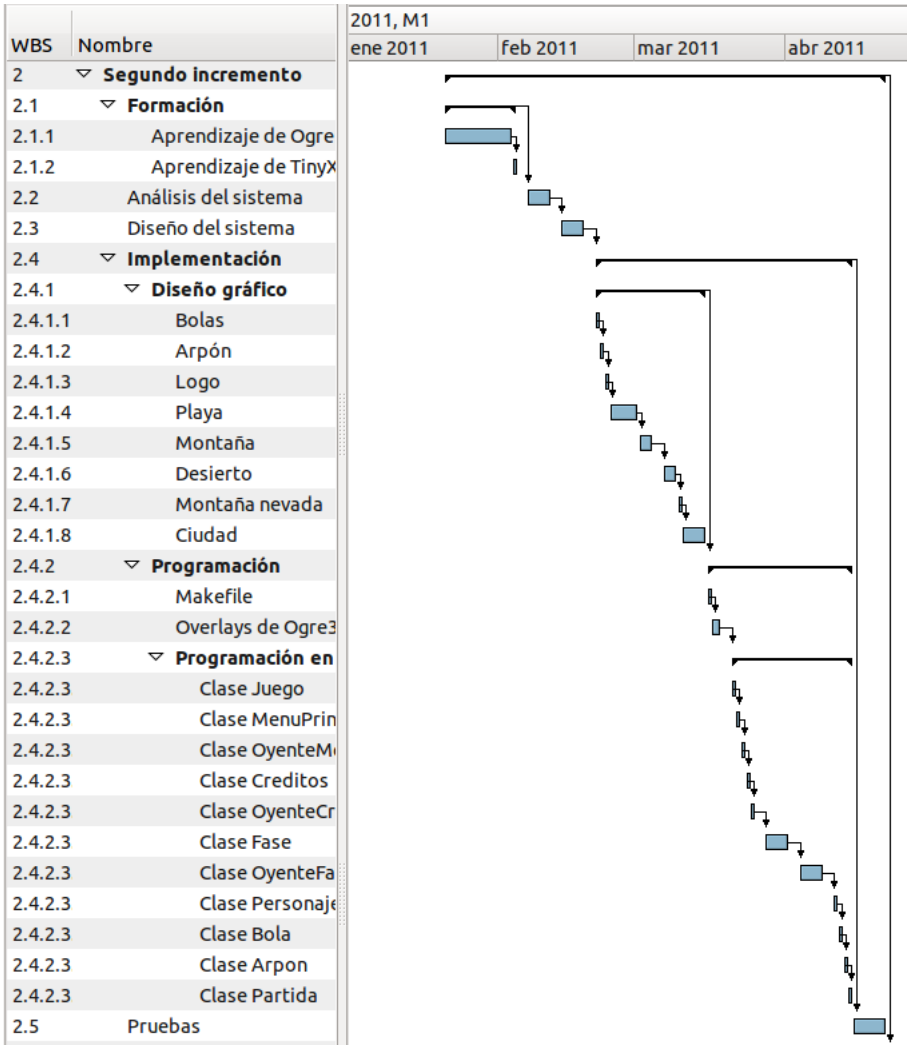

Figura 3.9: Diagrama de Gantt - Segundo incremento

WBS	Nombre	Trabajo
3	Tercer incremento	55d
3.1	Formación	6d
3.1.1	Aprendizaje de Ogre 3D	2d
3.1.2	Aprendizaje de SDL y SDL mixer	3d
3.1.3	Aprendizaje de Audacity	1d
3.2	Análisis del sistema	5d
3.3	Diseño del sistema	5d
3.4	Implementación	34d
3.4.1	Diseño gráfico	7d
3.4.1.1	Ermitaño	3d
3.4.1.2	Carpintero	3d
3.4.1.3	Partículas	1d
3.4.2	Sonido y música	9d
3.4.2.1	Búsqueda de sonidos	2d
3.4.2.2	Búsqueda de música	5d
3.4.2.3	Grabación de voces	1d
3.4.2.4	Edición con Audacity	1d
3.4.3	Programación	18d
3.4.3.1	Scripts de partículas de Ogre	2d
3.4.3.2	Niveles XML	4d
3.4.3.3	Programación en C++	12d
3.4.3.3.1	Clase Cubo	1d
3.4.3.3.2	Clase Carpintero	1d
3.4.3.3.3	Clase Ermitaño	1d
3.4.3.3.4	Clase Fase (ampliación)	2d
3.4.3.3.5	Clase OyenteFase (ampliación)	2d
3.4.3.3.6	Clase Partida (ampliación)	1d
3.4.3.3.7	Clase SistemaParticulas	1d
3.4.3.3.8	Clase Sonido	3d
3.5	Pruebas	5d

Tabla 3.3: Tareas del tercer incremento

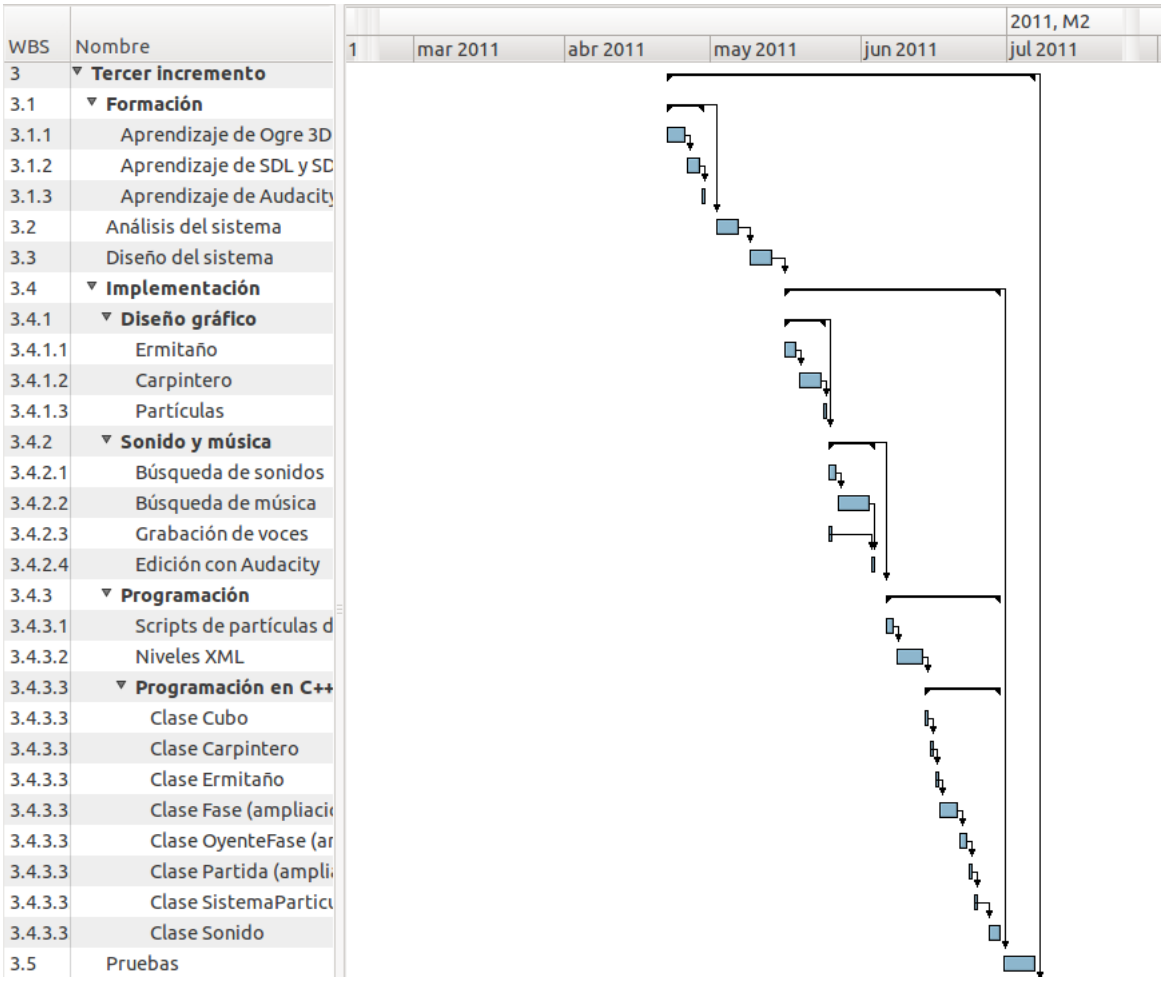

Figura 3.10: Diagrama de Gantt - Tercer incremento

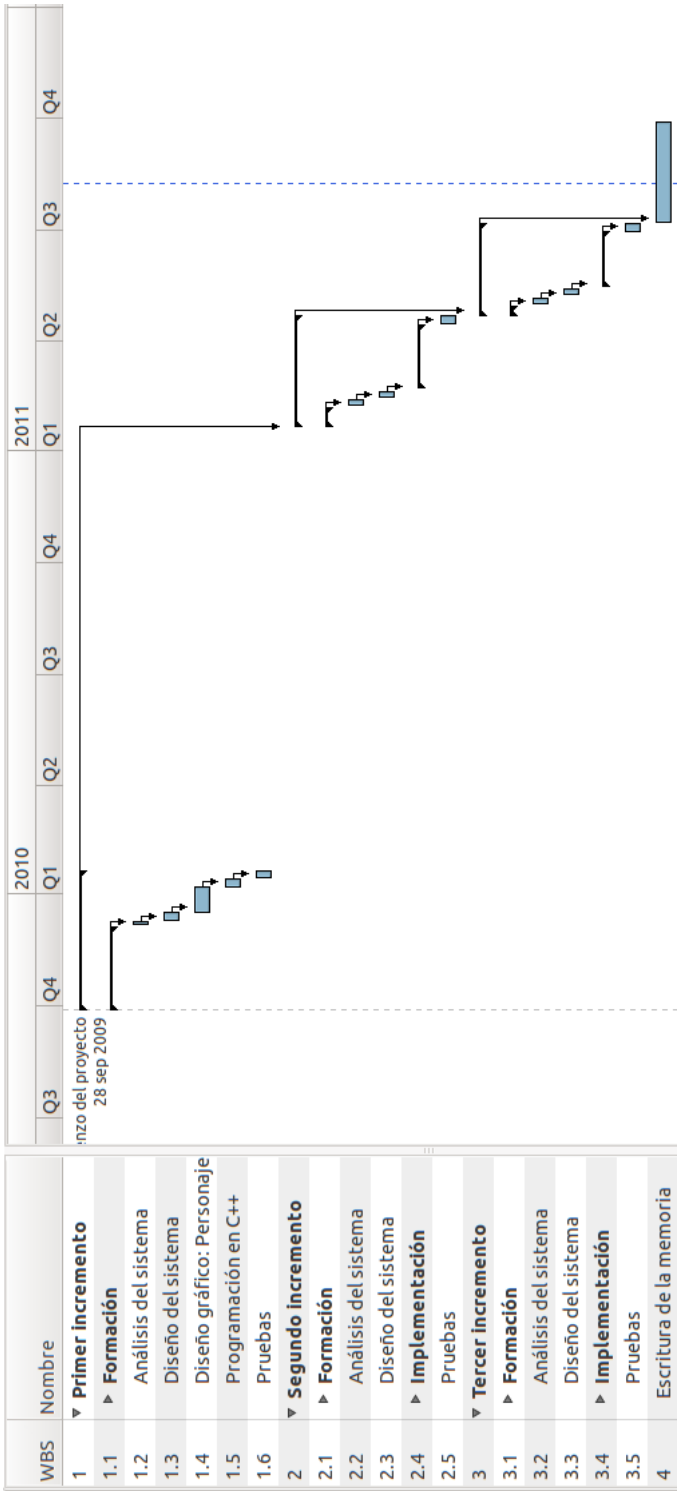

Figura 3.11: Diagrama de Gantt - Proyecto completo

Capítulo 4

Análisis

4.1. Funcionalidades del sistema

A continuación se describen los elementos básicos del juego y cómo se ha enfocado su implementación:

4.1.1. Juego

El juego en sí consta de varios objetos, existirá una clase controladora llamada Juego.

Un objeto juego del tipo Juego se creará al comenzar la aplicación. Esta es la clase principal de Balloon Breakers. Se encargará de crear todos los objetos necesarios en cada momento así como de cambiar de escena pasando del menú principal al juego o a los créditos.

4.1.2. Menú principal

En el menú principal el jugador podrá iniciar una partida, ver los créditos del juego o salir.

Cuando la aplicación comienza y el objeto juego es creado este se encarga de crear un objeto tipo MenuPrincipal y de cambiar el estado del juego a MenuPrincipal. La clase MenuPrincipal contiene los métodos y atributos necesarios para la gestión del menú principal.

4.1.3. Fase

El juego se divide en fases. Sin embargo, sólo una fase es jugada en un momento dado por lo cual es sólo necesario tener en cuenta la fase actual ya que el resto se cargará en un futuro. Esto se consigue mediante la implementación de la clase Fase cuyos atributos cambiarán dependiendo de la fase que el jugador esté jugando.

Las fases del juego son implementadas mediante la clase Fase. Esta clase contiene referencias a otras clases que la propia clase Fase crea: Personaje, Bola, Arpon... Existe una función que inicializa los atributos de la fase leyéndolos desde ficheros XML.

4.2. Modelo de casos de uso

A continuación se describen los casos de usos que especifican el comportamiento deseado del sistema.

He dividido los casos de uso según el ámbito dónde se utilizan: durante el menú principal, los créditos o jugando una fase.

4.2.1. Menú principal

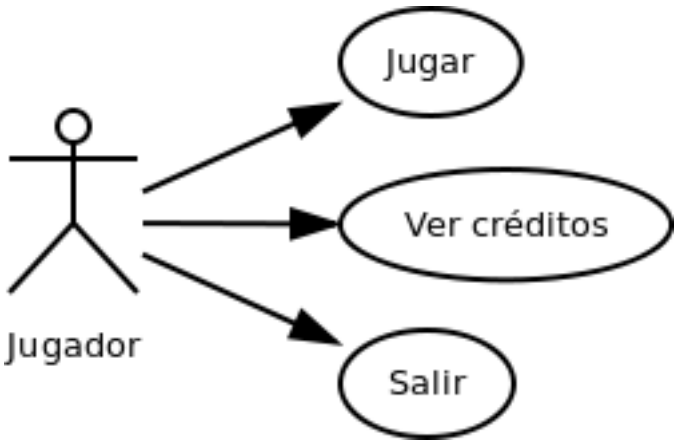

Figura 4.1: Diagrama de casos de uso: Menú principal

Caso de uso: Jugar.
Actor principal: Jugador.
Precondiciones: el juego está en el menú principal.
Postcondiciones: se comienza una nueva partida.
Escenario principal:
1. El jugador selecciona la opción “Play”.
2. Se sale del menú principal.
3. Se comienza una nueva partida desde la fase primera.
Escenario alternativos o de error:
* El usuario cierra la aplicación.

Caso de uso: Ver créditos.
Actor principal: Jugador.
Precondiciones: el juego está en el menú principal.
Postcondiciones: se entra en la pantalla con los créditos del juego.
Escenario principal:
1. El jugador selecciona la opción “Credits”.
2. Se sale del menú principal.
3. Se muestra la pantalla de créditos.
Escenario alternativos o de error:
* El usuario cierra la aplicación.

Caso de uso: Salir.
Actor principal: Jugador.
Precondiciones: el juego está en el menú principal.
Postcondiciones: se sale del juego cerrando la aplicación.
Escenario principal:

1. El jugador selecciona la opción “Exit”.
2. Se sale del juego.
Escenario alternativos o de error: ninguno.

4.2.2. Jugando fase

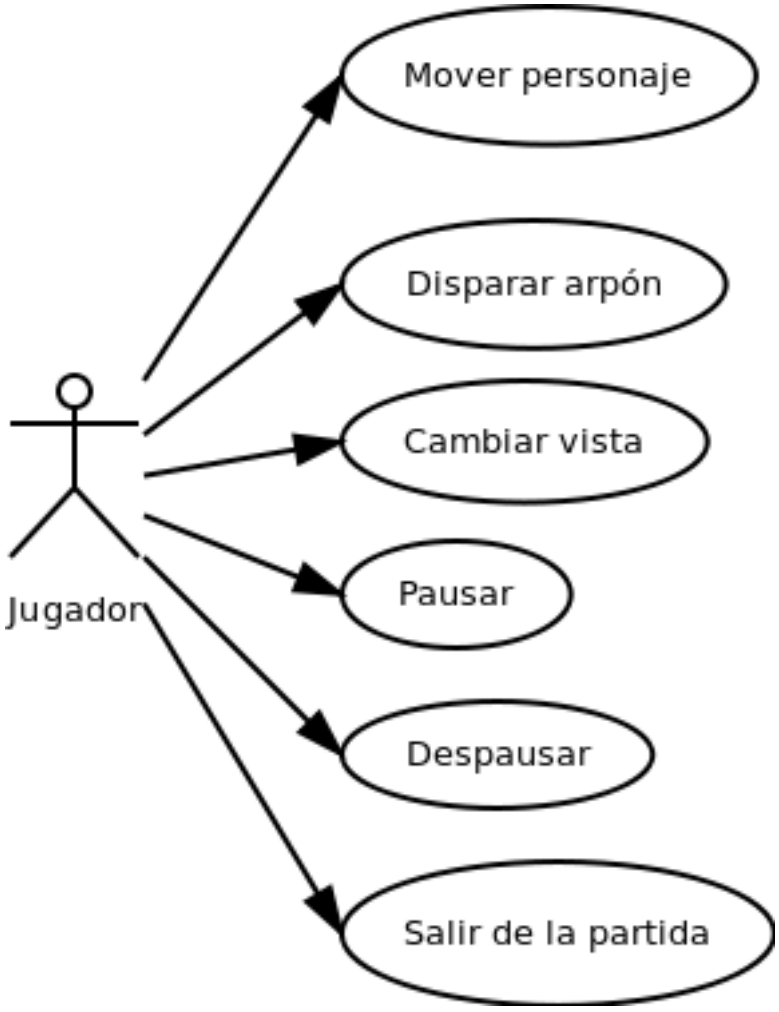

Figura 4.2: Diagrama de casos de uso: Jugando fase

Caso de uso: Mover personaje.
Actor principal: Jugador.
Precondiciones:
* El juego está en una fase.
* El juego no está pausado.
* El juego no está mostrando “Game Over”.
* El juego no está mostrando “Time Over”.
* El personaje no está siendo derribado.
* La fase no está introduciéndose.
* La fase no está acabando.

Postcondiciones: se mueve al personaje en la dirección indicada.
Escenario principal:
1. El jugador presiona una de las teclas de movimiento.
2. El sistema mueve el personaje en la dirección indicada.
Escenario alternativos o de error:
2a. El personaje no puede ser movido porque se encuentra al borde del escenario.
* El usuario cierra la aplicación.

Caso de uso: Disparar arpón.
Actor principal: Jugador.
Precondiciones:
* El juego está en una fase.
* El juego no está pausado.
* El juego no está mostrando “Game Over”.
* El juego no está mostrando “Time Over”.
* El personaje no está siendo derribado.
* La fase no está introduciéndose.
* La fase no está acabando.
Postcondiciones: se dispara un arpón desde la posición del personaje.
Escenario principal:
1. El jugador presiona la tecla de disparo.
2. El sistema crea el arpón que se mueve verticalmente hacia arriba.
Escenario alternativos o de error:
2a. No se crea ningún arpón puesto que el arpón ya existe.
* El usuario cierra la aplicación.

Caso de uso: Pausar.
Actor principal: Jugador.
Precondiciones:
* El juego está en una fase.
* El juego no está pausado.
* El juego no está mostrando “Game Over”.
* El juego no está mostrando “Time Over”.
* El personaje no está siendo derribado.
* La fase no está introduciéndose.
* La fase no está acabando.
Postcondiciones: se pausa el juego.
Escenario principal:
1. El jugador presiona la tecla de pausar/despausar.
2. El sistema pausa el juego mostrando el mensaje “Pause”.
Escenario alternativos o de error:
* El usuario cierra la aplicación.

Caso de uso: Despausar.
Actor principal: Jugador.
Precondiciones:
* El juego está en una fase.
* El juego está pausado.
Postcondiciones: se despausa el juego.

Escenario principal:
1. El jugador presiona la tecla de pausar/despausar.
2. El sistema despausa el juego ocultando el mensaje “Pause”.
Escenario alternativos o de error:
* El usuario cierra la aplicación.

Caso de uso: Salir de la partida.
Actor principal: Jugador.
Precondiciones:
* El juego está en una fase.
Postcondiciones: se sale de la partida volviendo al menú principal.
Escenario principal:
1. El jugador presiona la tecla de salir.
2. El sistema termina la partida volviendo al menú principal.
Escenario alternativos o de error:
* El usuario cierra la aplicación.

4.2.3. Créditos

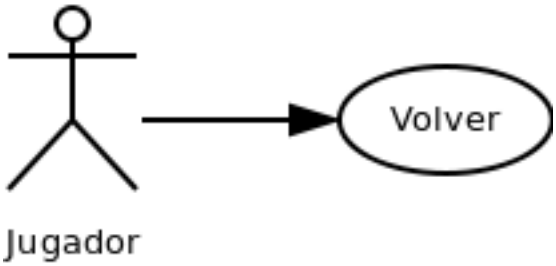

Figura 4.3: Diagrama de casos de uso: Créditos

Caso de uso: Volver.
Actor principal: Jugador.
Precondiciones: El juego está mostrando los créditos.
Postcondiciones: se sale de los créditos volviendo al menú principal.
Escenario principal:
1. El jugador presiona la tecla de volver.
2. El sistema vuelve al menú principal.
Escenario alternativos o de error:
* El usuario cierra la aplicación.

4.3. Modelo conceptual de datos

Para el desarrollo del juego necesitaremos, principalmente, las siguientes clases:

- **Juego:** Clase principal del juego. Se encarga de gestionar el estado del juego y de crear y controlar el resto de clases.
- **MenuPrincipal:** Contiene los métodos y atributos para el control del menú principal del juego desde donde se accede a las partes principales de este: jugar partida y ver créditos.
- **OyenteMenuPrincipal:** Contiene métodos y atributos para el control del teclado y demás eventos en el menú principal como el movimiento del fondo, el control de la sección seleccionada, etcétera.
- **Creditos:** Contiene los métodos y atributos para el control de los créditos.
- **OyenteCreditos:** Contiene métodos y atributos para el control del teclado y demás eventos en el menú principal como el control del teclado para detectar si el jugador pulsa la tecla espacio para volver al menú principal.
- **Fase:** Contiene los métodos y atributos para la gestión de las fases como la lectura desde archivos XML, el personaje, listas de bolas y demás elementos como cubos, animales, sistemas de partículas, etcétera.
- **OyenteFase:** Contiene métodos y atributos para el control del teclado y demás eventos en el menú principal como el control del teclado, gestión de colisiones, etcétera.
- **Sonido:** Contiene métodos y atributos para la gestión de la música, sonido y voces del juego.
- **Partida:** La clase Partida contiene los atributos y métodos necesarios para la gestión de una partida como son las vidas y la puntuación conseguidas.
- **Personaje:** Contiene métodos y atributos para la gestión del personaje. Dirección en que está mirando, posición, etcétera. Contiene punteros a clases de Ogre para representar el personaje.
- **Arpon:** Contiene métodos y atributos para la gestión del arpón que el personaje dispara, es decir, en que posición se encuentra y a que altura o si este no se ha disparado. Contiene punteros a clases de Ogre para representar el arpón.
- **Cubo:** Contiene métodos y atributos para la gestión de los cubos de las fases, es decir, las plataformas que el personaje puede destruir con el arpón y en las cuales las bolas colisionan. Contiene punteros a clases de Ogre para representar dichos cubos.
- **Carpintero:** Contiene métodos y atributos para la gestión de los pájaros carpinteros que aparecen en las fases como son el control de la posición y dirección en que el carpintero vuela. Contiene también punteros a las clases de SceneManager y Entity de Ogre para representar los carpinteros.
- **Ermitano:** Similarmente a la clase Carpintero esta clase contiene los métodos y atributos para la gestión de los cangrejos ermitaños que aparecen durante el juego.
- **Bola:** Esta clase contiene métodos y atributos para gestionar las bolas del juego, su tamaño, velocidad, posición y dirección a donde se mueven. También contiene referencias a las clases de Ogre necesarias para representar las bolas.
- **SistemaParticulas:** Esta clase contiene métodos y atributos para gestionar los efectos de partículas de Ogre que se muestran en la fase cuando se destruye una bola o un animal.

A continuación se incluye el diagrama de clases conceptual que contiene dichas clases. Se han omitido las clases implementadas en Ogre, SDL y TinyXML:

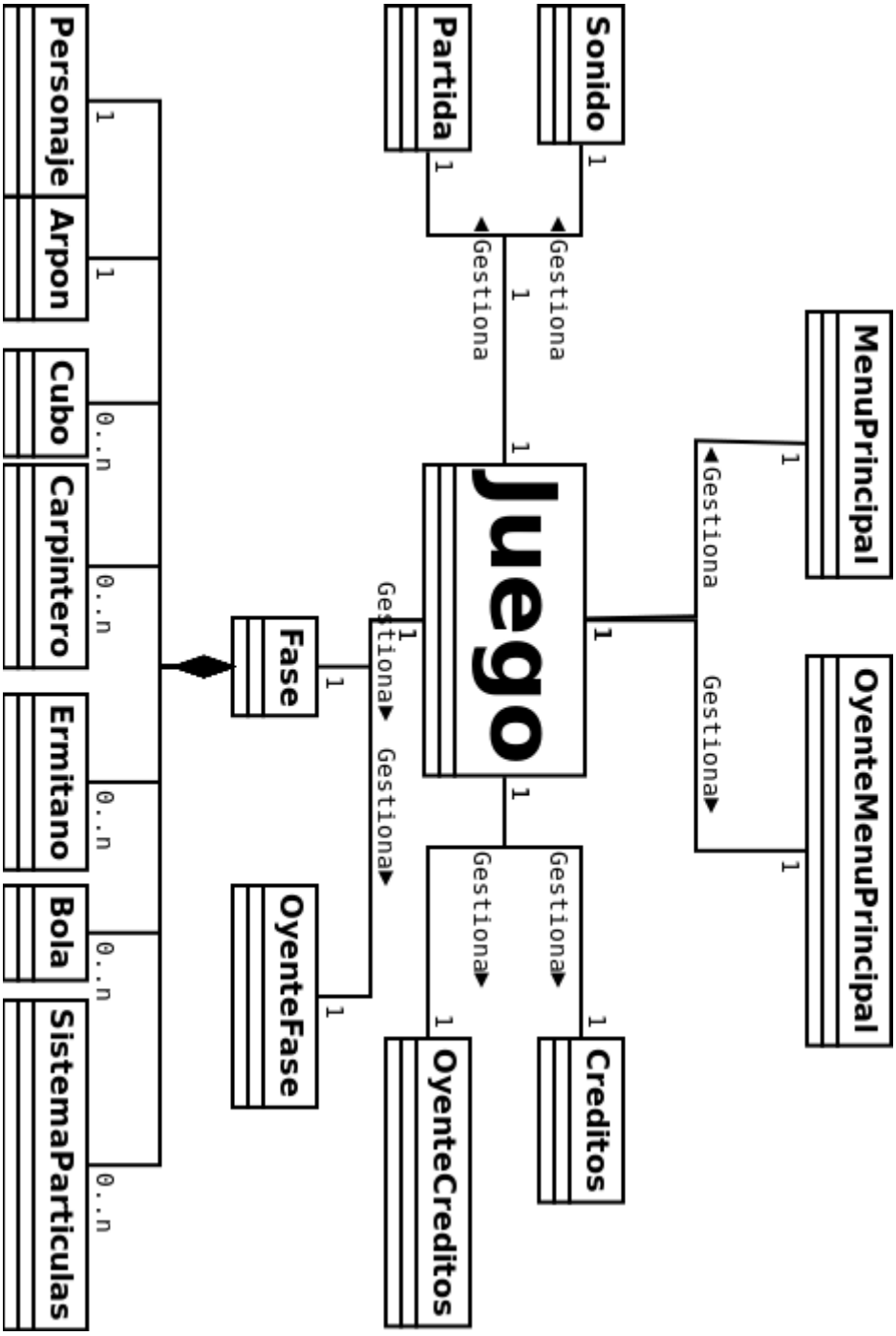

Figura 4.4: Diagrama de clases conceptual

4.4. Modelo de comportamiento

A continuación se incluye el modelo de comportamiento que especifica cómo debe de actuar el sistema. Este modelo consta de dos partes:

- **Diagramas de secuencias del sistema:** muestran la secuencia de eventos entre los actores y el

sistema.

- **Contratos de las operaciones del sistema:** describen el efecto que producen las operaciones del sistema.

Al igual que en el punto anterior he dividido el modelo de comportamiento en tres secciones.
Hago referencia a diferentes estados de los objetos. En la siguiente subsección se mostrará un diagrama de estados para los objetos con más estados diferentes del juego: Personaje, Juego y Fase

4.4.1. Menú principal

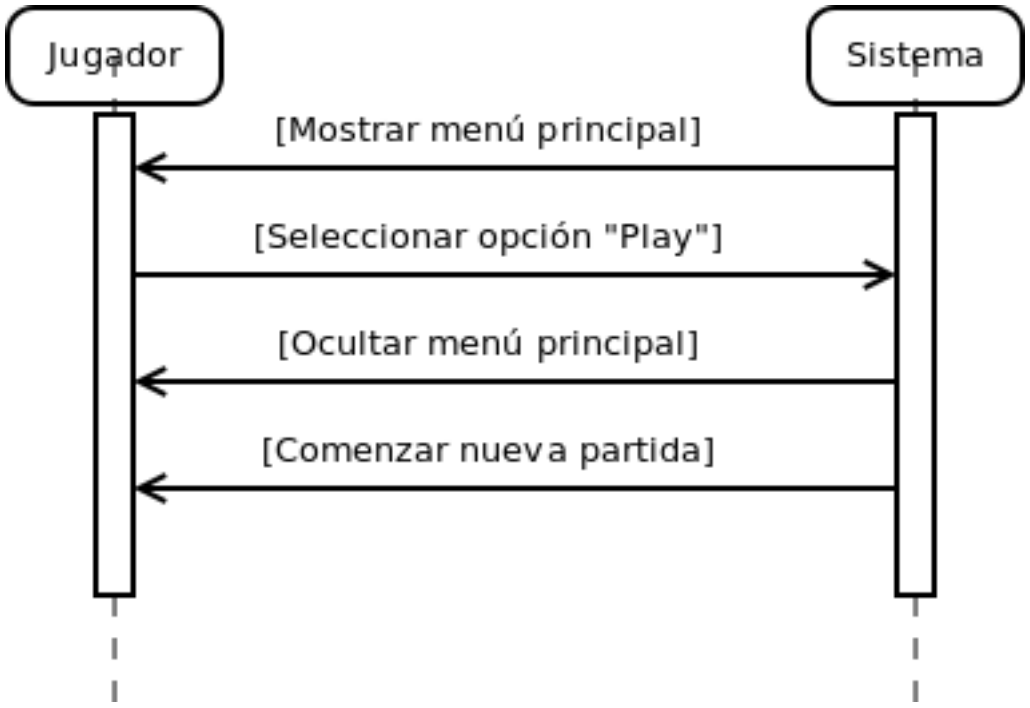

Figura 4.5: Diagrama de secuencia del sistema: Jugar

Contrato de las operaciones

Operación: Mostrar menú principal.
Responsabilidad: muestra el menú principal con las opciones “Play”, “Credits” y “Exit”.
Precondiciones: ninguna.
Postcondiciones: se ha cargado las imágenes, capas, sonido y música para el menú principal.

Operación: Seleccionar opción “Play”.
Responsabilidad: marca la opción “Play” como seleccionada y emite un sonido de selección.
Precondiciones: se ha mostrado el menú principal.
Postcondiciones: se ha marcado la opción “Play” y se ha emitido un sonido de selección.

Operación: Ocultar menú principal.
Responsabilidad: oculta el menú principal para mostrar otra escena.
Precondiciones: se ha seleccionado la opción “Play”.

Postcondiciones: se ha cargado las imágenes, capas, sonido y música para el menú principal.

Operación: Comenzar nueva partida.
Responsabilidad: comienza una partida del juego.
Precondiciones: ninguna.
Postcondiciones: se ha creado una instancia partida de Partida. Se ha pasado el control a OyenteFase. Se ha cargado la primera fase del juego y se ha comenzado la partida cambiando el estado del objeto juego a PARTIDA.

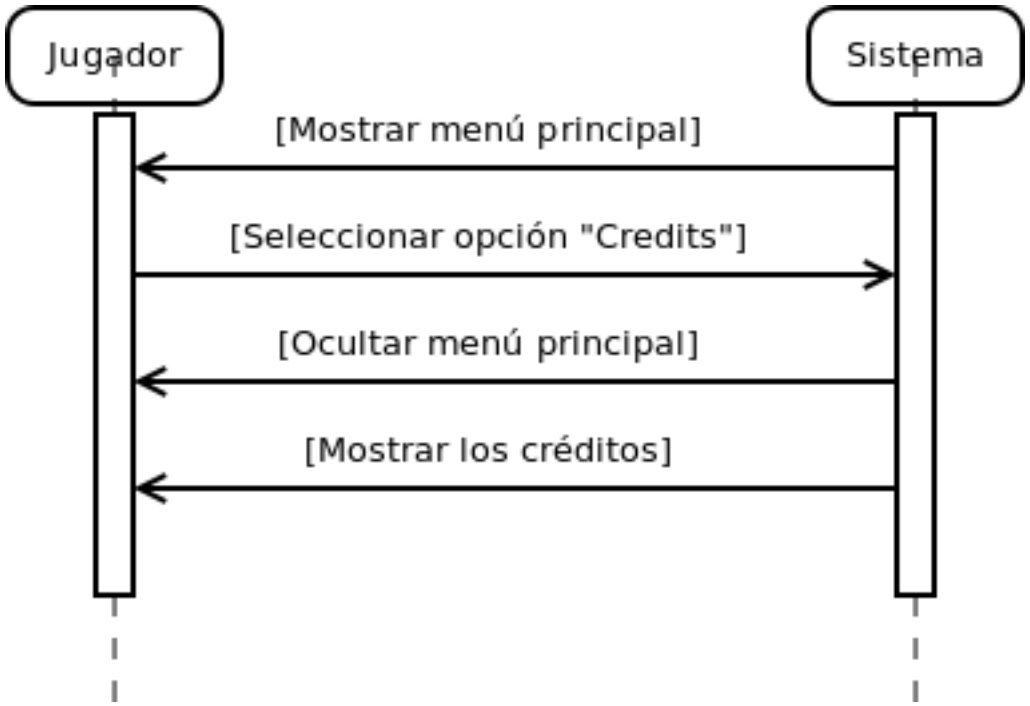

Figura 4.6: Diagrama de secuencia del sistema: Ver créditos

Contrato de las operaciones

Operación: Mostrar menú principal.
Responsabilidad: muestra el menú principal con las opciones “Play”, “Credits” y “Exit”.
Precondiciones: ninguna.
Postcondiciones: se ha cargado las imágenes, capas, sonido y música para el menú principal.

Operación: Seleccionar opción “Credits”.
Responsabilidad: marca la opción “Credits” como seleccionada y emite un sonido de selección.
Precondiciones: se ha mostrado el menú principal.
Postcondiciones: se ha marcado la opción “Credits” y se ha emitido un sonido de selección.

Operación: Ocultar menú principal.
Responsabilidad: oculta el menú principal para mostrar otra escena.
Precondiciones: se ha seleccionado la opción “Credits”.
Postcondiciones: se ha cargado las imágenes, capas, sonido y música para el menú principal.

Operación: Mostrar los créditos.
Responsabilidad: muestra los créditos del juego.
Precondiciones: ninguna.
Postcondiciones: se ha pasado el control a OyenteCreditos. Se ha cargado la pantalla de créditos. Se ha cambiando el estado del objeto juego a CREDITOS.

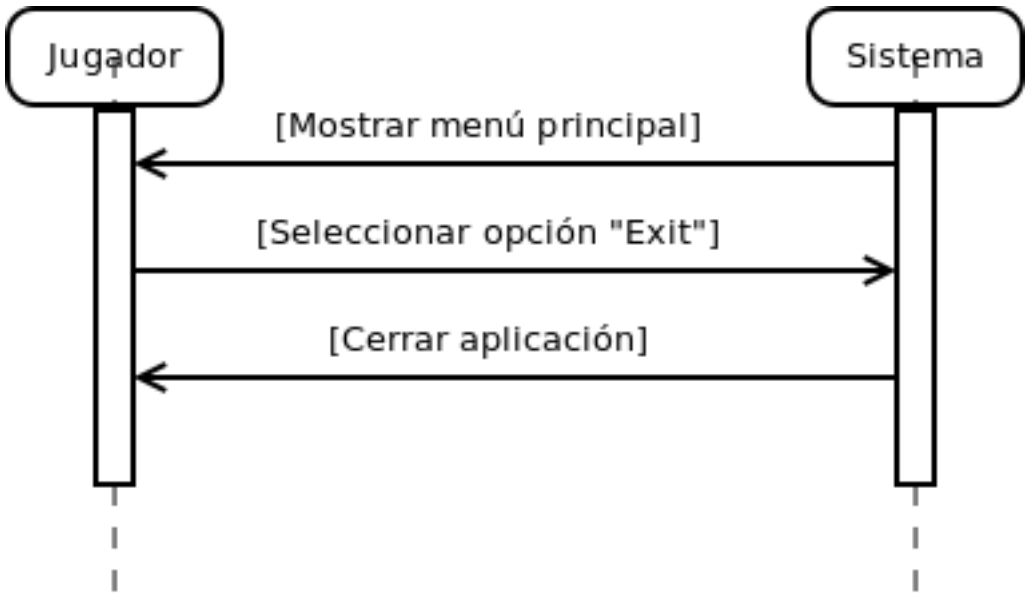

Figura 4.7: Diagrama de secuencia del sistema: Salir

Contrato de las operaciones

Operación: Mostrar menú principal.
Responsabilidad: muestra el menú principal con las opciones “Play”, “Credits” y “Exit”.
Precondiciones: ninguna.
Postcondiciones: se ha cargado las imágenes, capas, sonido y música para el menú principal.

Operación: Seleccionar opción “Exit”.
Responsabilidad: marca la opción “Exit” como seleccionada y emite un sonido de selección.
Precondiciones: se ha mostrado el menú principal.
Postcondiciones: se ha marcado la opción “Exit” y se ha emitido un sonido de selección.

Operación: Cerrar aplicación.
Responsabilidad: cierra la aplicación liberando la memoria.
Precondiciones: se ha seleccionado la opción “Exit”.
Postcondiciones: se han destruido los objetos creados. Se ha cerrado la aplicación.

4.4.2. Jugando fase

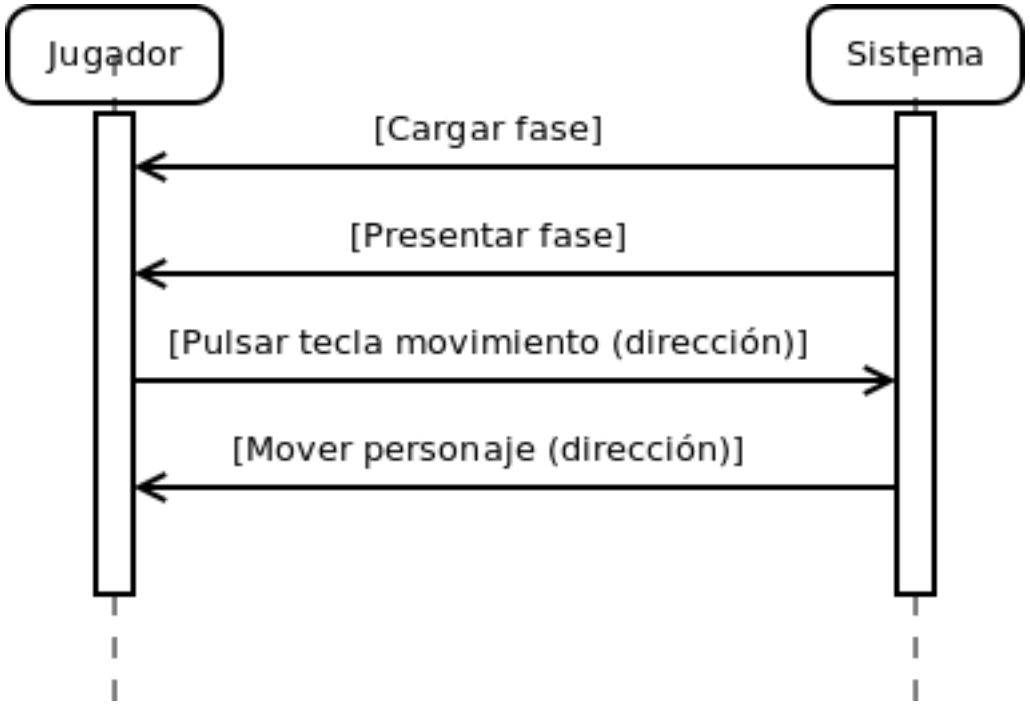

Figura 4.8: Diagrama de secuencia del sistema: Mover personaje

Contrato de las operaciones

Operación: Cargar fase.
Responsabilidad: lee la fase de un fichero XML y la carga en memoria.
Precondiciones: el objeto juego se encuentra en el estado PARTIDA. existe el fichero XML.
Postcondiciones: se ha leído el fichero XML y creado los objetos necesarios según los datos contenidos en el fichero XML. El objeto fase pasa a tener el estado INTRODUCCION.

Operación: Presentar fase.
Responsabilidad: muestra los mensajes “Ready?” y “Go!” emitiendo el sonido correspondientes.
Precondiciones: se ha cargado la fase.
Postcondiciones: se ha mostrado la presentación de la fase y se ha dado al jugador el control del personaje. El objeto fase pasa a tener el estado JUGANDO.

Operación: Pulsar tecla movimiento (dirección).
Responsabilidad: se lee una de las teclas (arriba, abajo, izquierda o derecha) para posteriormente mover el personaje en dicha dirección.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha leído la dirección en la que el jugador desea mover el personaje.

Operación: Mover personaje (dirección).
Responsabilidad: mueve al personaje en la dirección indicada.
Precondiciones: se ha pulsado una tecla de movimiento.
Postcondiciones: se ha movido al personaje en la dirección indicada una cantidad proporcional al tiempo

transcurrido tras el último frame. El objeto personaje ha cambiado su estado a CORRIENDO.

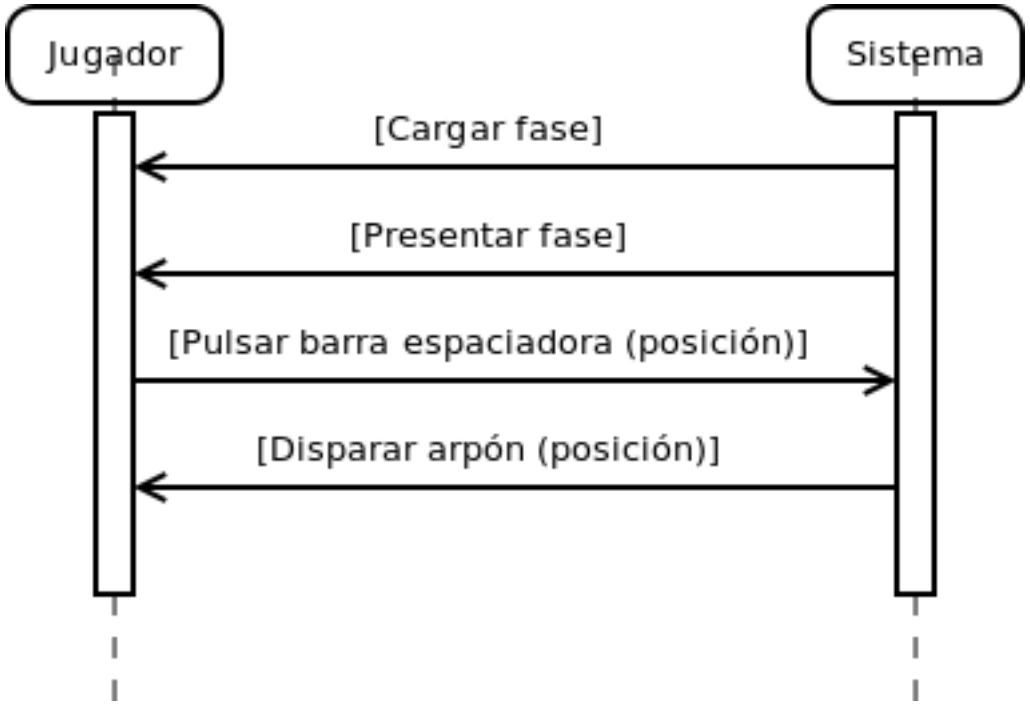

Figura 4.9: Diagrama de secuencia del sistema: Disparar arpón

Contrato de las operaciones

Operación: Cargar fase.
Responsabilidad: lee la fase de un fichero XML y la carga en memoria.
Precondiciones: el objeto juego se encuentra en el estado PARTIDA. existe el fichero XML.
Postcondiciones: se ha leído el fichero XML y creado los objetos necesarios según los datos contenidos en el fichero XML. El objeto fase pasa a tener el estado INTRODUCCION.

Operación: Presentar fase.
Responsabilidad: muestra los mensajes “Ready?” y “Go!” emitiendo el sonido correspondientes.
Precondiciones: se ha cargado la fase.
Postcondiciones: se ha mostrado la presentación de la fase y se ha dado al jugador el control del personaje. El objeto fase pasa a tener el estado JUGANDO.

Operación: Pulsar barra espaciadora (posición).
Responsabilidad: se detecta la pulsación de la tecla espacio y la posición del personaje.
Precondiciones: se ha presentado la fase. El arpón no está siendo disparado.
Postcondiciones: se ha detectado la pulsación de la tecla espacio y se ha leído la posición en la que el personaje se encuentra. El objeto personaje ha cambiado su estado a DISPARANDO.

Operación: Disparar arpón (posición).
Responsabilidad: dispara el arpón desde la posición del personaje.
Precondiciones: se ha pulsado una tecla espacio y no existe un arpón visible.
Postcondiciones: se ha creado el arpón en la posición del personaje que se mueve ascendentemente. El

objeto personaje ha cambiado su estado a REPOSO.

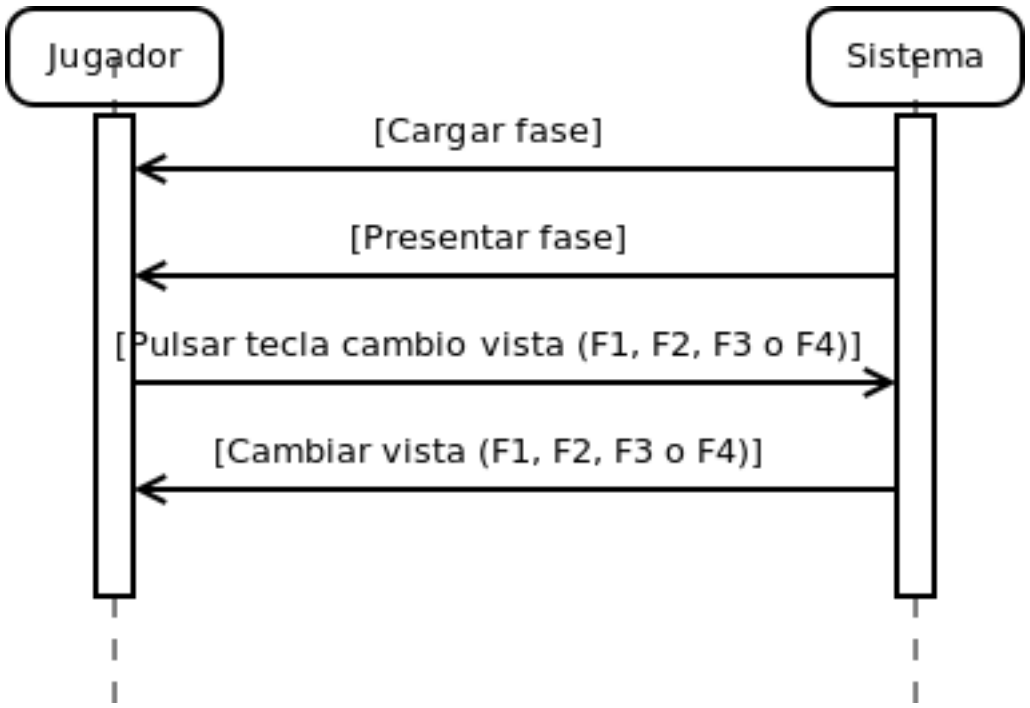

Figura 4.10: Diagrama de secuencia del sistema: Cambiar vista

Contrato de las operaciones

Operación: Cargar fase.
Responsabilidad: lee la fase de un fichero XML y la carga en memoria.
Precondiciones: el objeto juego se encuentra en el estado PARTIDA. existe el fichero XML.
Postcondiciones: se ha leído el fichero XML y creado los objetos necesarios según los datos contenidos en el fichero XML. El objeto fase pasa a tener el estado INTRODUCCION.

Operación: Presentar fase.
Responsabilidad: muestra los mensajes “Ready?” y “Go!” emitiendo el sonido correspondientes.
Precondiciones: se ha cargado la fase.
Postcondiciones: se ha mostrado la presentación de la fase y se ha dado al jugador el control del personaje. El objeto fase pasa a tener el estado JUGANDO.

Operación: Pulsar tecla cambio vista.
Responsabilidad: se detecta la pulsación de una de las teclas F1, F2, F3 o F4.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha detectado la pulsación una de las teclas F1, F2, F3 o F4.

Operación: Cambiar vista.
Responsabilidad: se cambia la el punto de vista de la fase.
Precondiciones: se ha presentado la fase y el objeto fase esta en estado JUGANDO.
Postcondiciones: se ha cambiado el punto de vista de la fase.

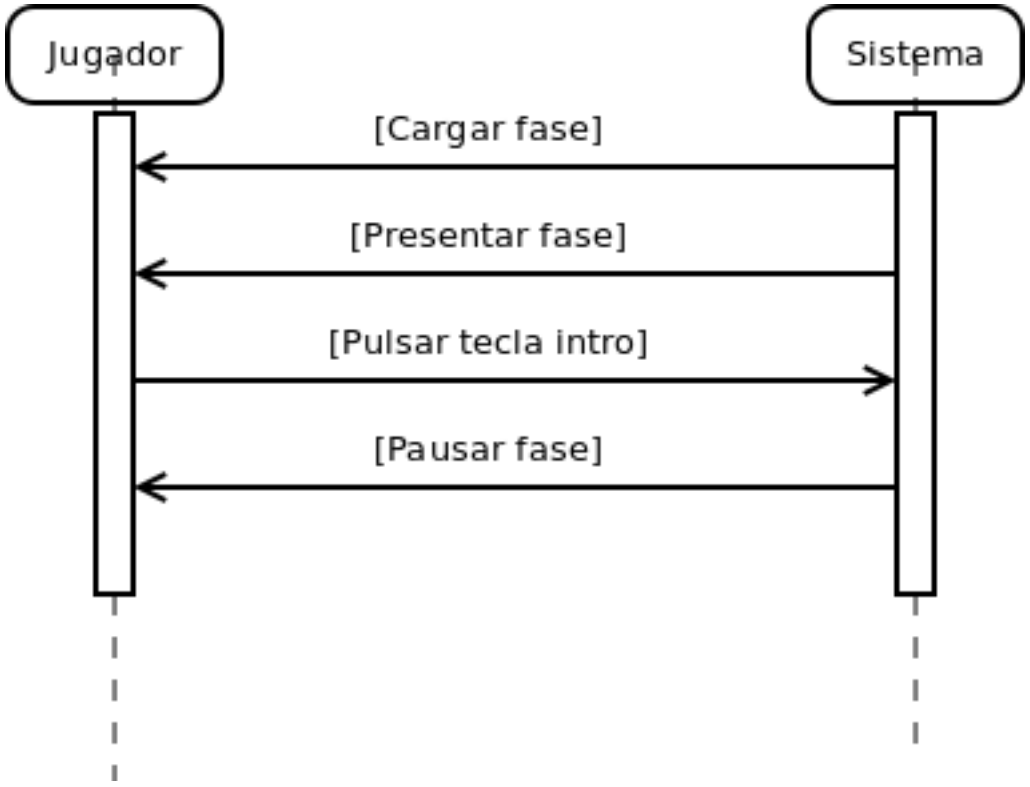

Figura 4.11: Diagrama de secuencia del sistema: Pausar

Contrato de las operaciones

Operación: Cargar fase.
Responsabilidad: lee la fase de un fichero XML y la carga en memoria.
Precondiciones: el objeto juego se encuentra en el estado PARTIDA. existe el fichero XML.
Postcondiciones: se ha leído el fichero XML y creado los objetos necesarios según los datos contenidos en el fichero XML. El objeto fase pasa a tener el estado INTRODUCCION.

Operación: Presentar fase.
Responsabilidad: muestra los mensajes “Ready?” y “Go!” emitiendo el sonido correspondientes.
Precondiciones: se ha cargado la fase.
Postcondiciones: se ha mostrado la presentación de la fase y se ha dado al jugador el control del personaje. El objeto fase pasa a tener el estado JUGANDO.

Operación: Pulsar tecla intro.
Responsabilidad: se detecta la pulsación de la tecla intro.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha detectado la pulsación de la tecla intro.

Operación: Pausar fase.
Responsabilidad: se pausa la fase.
Precondiciones: se ha presentado la fase y el objeto fase no esta en estado PAUSA.
Postcondiciones: se ha cambiado el estado del objeto fase a PAUSA.

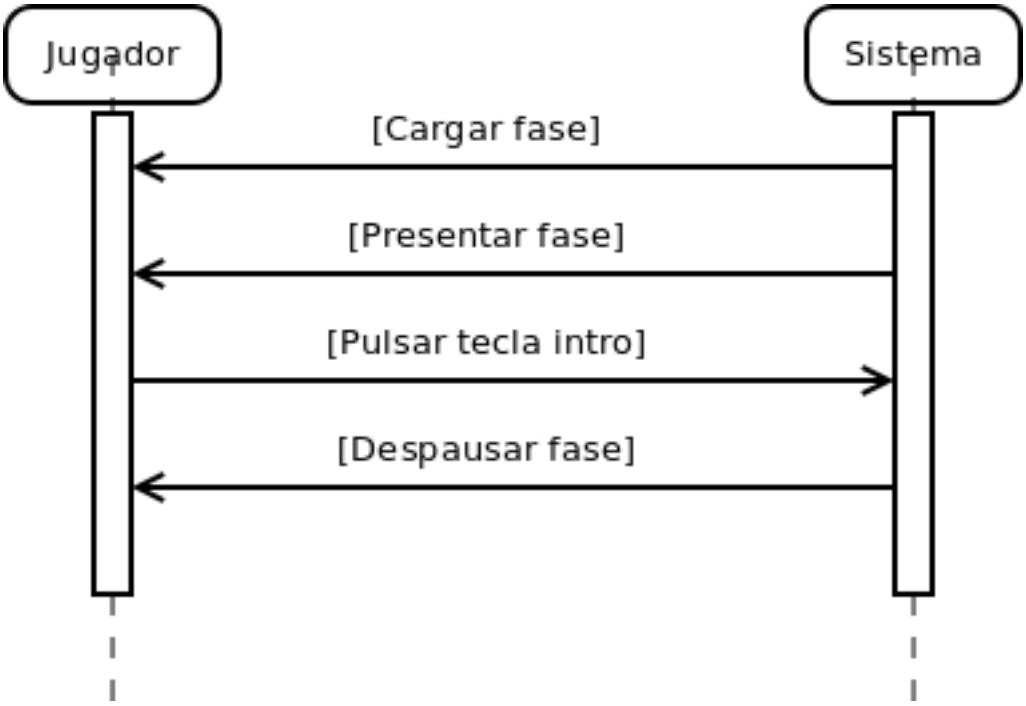

Figura 4.12: Diagrama de secuencia del sistema: Despausar

Contrato de las operaciones

Operación: Cargar fase.
Responsabilidad: lee la fase de un fichero XML y la carga en memoria.
Precondiciones: el objeto juego se encuentra en el estado PARTIDA. existe el fichero XML.
Postcondiciones: se ha leído el fichero XML y creado los objetos necesarios según los datos contenidos en el fichero XML. El objeto fase pasa a tener el estado INTRODUCCION.

Operación: Presentar fase.
Responsabilidad: muestra los mensajes “Ready?” y “Go!” emitiendo el sonido correspondientes.
Precondiciones: se ha cargado la fase.
Postcondiciones: se ha mostrado la presentación de la fase y se ha dado al jugador el control del personaje. El objeto fase pasa a tener el estado JUGANDO.

Operación: Pulsar tecla intro.
Responsabilidad: se detecta la pulsación de la tecla intro.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha detectado la pulsación de la tecla intro.

Operación: Despausar fase.
Responsabilidad: se pausa la fase.
Precondiciones: se ha presentado la fase y la fase está en estado PAUSA.
Postcondiciones: se pausa la fase pasando esta al estado PAUSA.

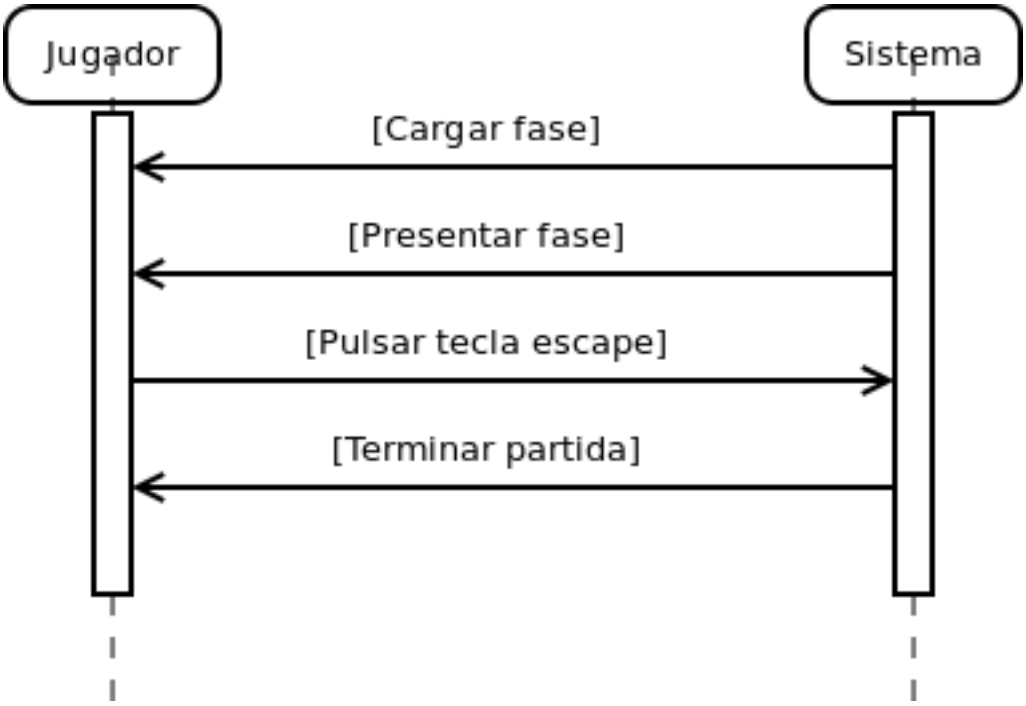

Figura 4.13: Diagrama de secuencia del sistema: Salir partida

Contrato de las operaciones

Operación: Cargar fase.
Responsabilidad: lee la fase de un fichero XML y la carga en memoria.
Precondiciones: el objeto juego se encuentra en el estado PARTIDA. existe el fichero XML.
Postcondiciones: se ha leído el fichero XML y creado los objetos necesarios según los datos contenidos en el fichero XML. El objeto fase pasa a tener el estado INTRODUCCION.

Operación: Presentar fase.
Responsabilidad: muestra los mensajes “Ready?” y “Go!” emitiendo el sonido correspondientes.
Precondiciones: se ha cargado la fase.
Postcondiciones: se ha mostrado la presentación de la fase y se ha dado al jugador el control del personaje. El objeto fase pasa a tener el estado JUGANDO.

Operación: Pulsar tecla escape.
Responsabilidad: se detecta la pulsación de la tecla escape.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha detectado la pulsación de la tecla escape.

Operación: Terminar partida.
Responsabilidad: se pausa la fase.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha salido de la partida. Se ha cambiado el estado del objeto juego a MENUPRINCIPAL.

4.4.3. Créditos

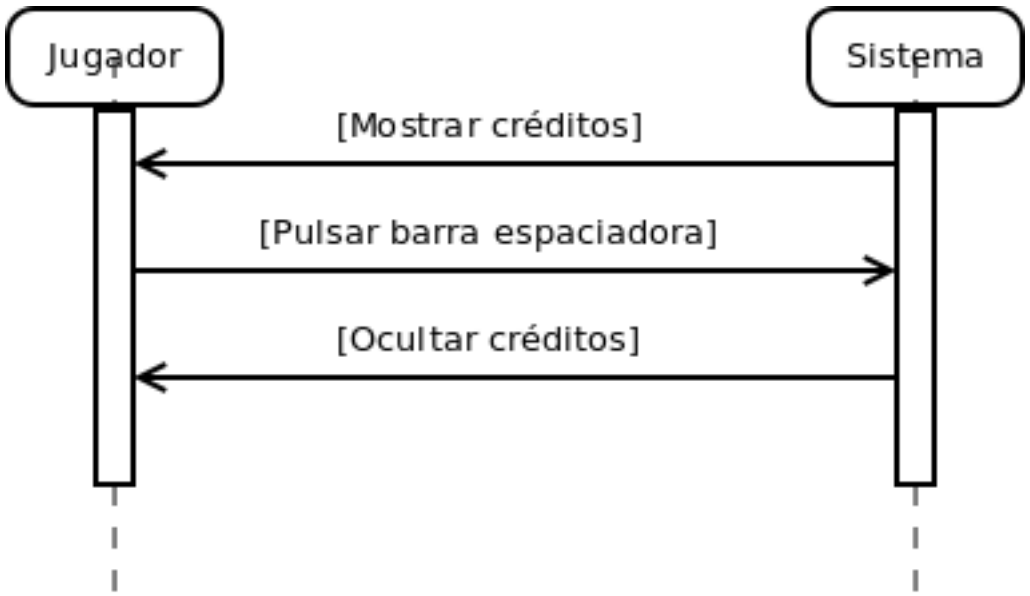

Figura 4.14: Diagrama de secuencia del sistema: Volver

Contrato de las operaciones

Operación: Cargar fase.
Responsabilidad: lee la fase de un fichero XML y la carga en memoria.
Precondiciones: el objeto juego se encuentra en el estado PARTIDA. existe el fichero XML.
Postcondiciones: se ha leído el fichero XML y creado los objetos necesarios según los datos contenidos en el fichero XML. El objeto fase pasa a tener el estado INTRODUCCION.

Operación: Presentar fase.
Responsabilidad: muestra los mensajes “Ready?” y “Go!” emitiendo el sonido correspondientes.
Precondiciones: se ha cargado la fase.
Postcondiciones: se ha mostrado la presentación de la fase y se ha dado al jugador el control del personaje. El objeto fase pasa a tener el estado JUGANDO.

Operación: Pulsar tecla escape.
Responsabilidad: se detecta la pulsación de la tecla escape.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha detectado la pulsación de la tecla escape.

Operación: Terminar partida.
Responsabilidad: se pausa la fase.
Precondiciones: se ha presentado la fase.
Postcondiciones: se ha salido de la partida. Se ha cambiado el estado del objeto juego a MENUPRINCIPAL.

4.5. Diagramas de estado del sistema

Dado el elevado número de estados y el complejo sistema de cambios de estos en los objetos juego y fase del juego he elaborado los siguientes diagramas de estado que describen el cambio de estados de dichos objetos y las condiciones que se tienen que dar para que dicho cambio se produzca.

Los círculos representan los diferentes estados y las líneas el cambio de un estado a otro. Junto a cada línea se incluye un resumen de las condiciones que se deben dar para que se produzca el cambio de estado.

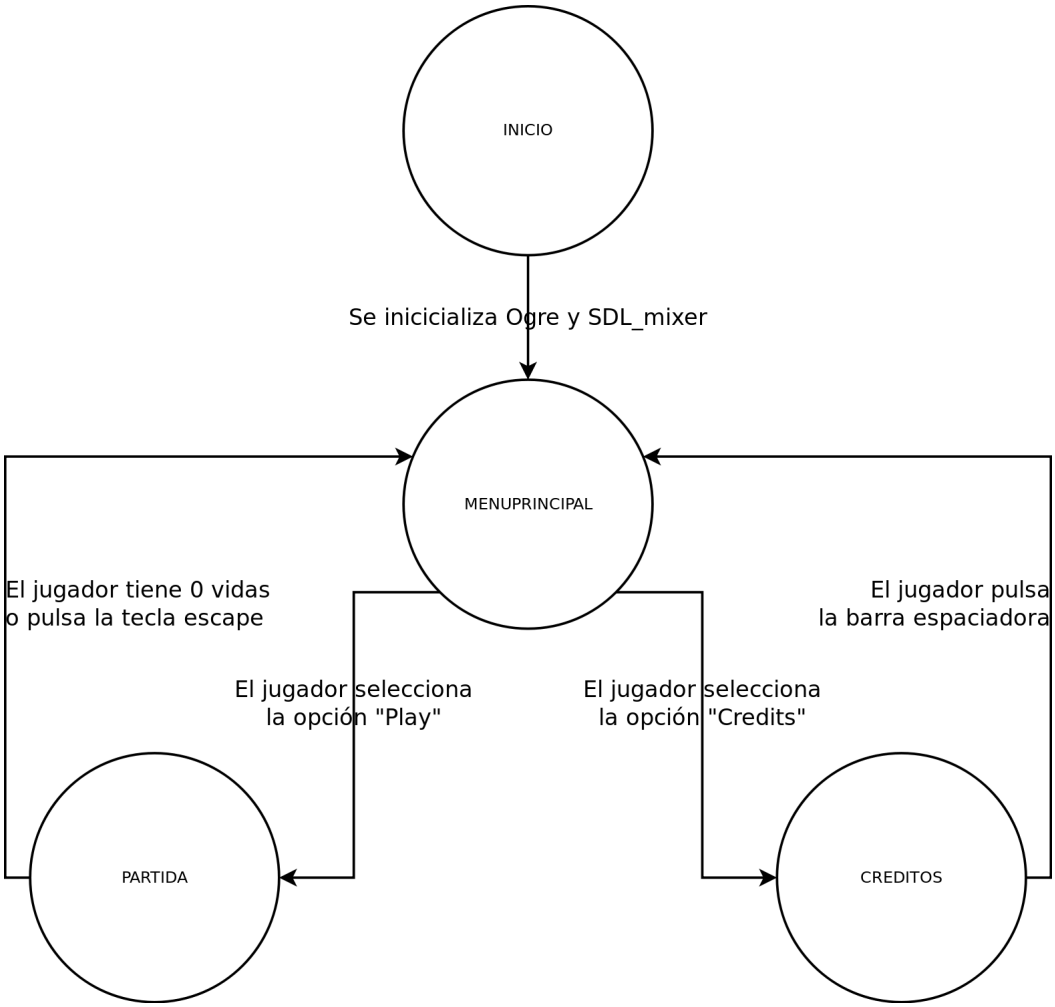

Figura 4.15: Diagrama de estados de la clase Juego

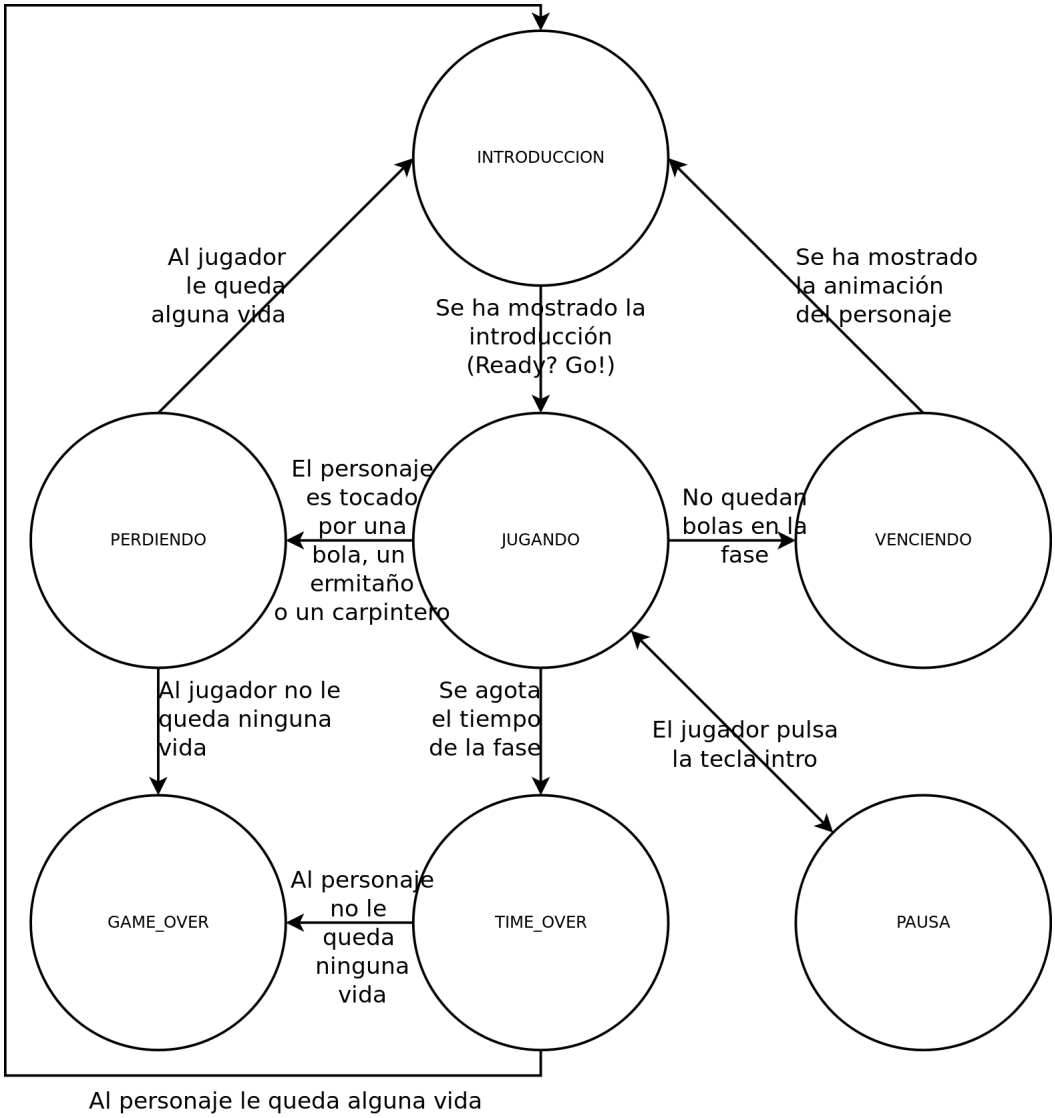

Figura 4.16: Diagrama de estados de la clase Fase

Capítulo 5

Diseño

5.1. Diseño del sistema

Una vez especificado qué hace el sistema en los capítulos anteriores continuamos con el diseño de este.

Se incluye a continuación el diagrama de clases de diseño y, posteriormente, se describe el comportamiento de cada una de las clases que lo componen.

5.1.1. Diagrama de clases de diseño

A continuación se incluye el diagrama de clases de diseño. Aquí se incluyen los atributos y las operaciones de las clases ampliando el capítulo anterior.

Figura 5.1: Diagrama de clases de diseño

5.1.2. Descripción de las clases

En este apartado describo el diseño de cada clase. Se incluyen las clases en orden alfabético.

Clase Arpon

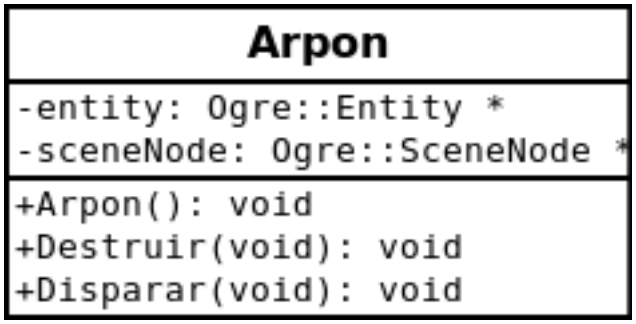

Figura 5.2: Clase Arpon

La clase Arpon describe el comportamiento y cualidades del arpón que dispara el personaje del juego. Contiene los siguientes atributos y métodos:

Atributos:

- **entity**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D (modelo, animación, texturas, materiales...).
- **sceneNode**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D entre otras.

Métodos:

- **Arpon()**: constructor de la clase, no recibe ningún parámetro.
- **Destruir()**: éste método se encarga de hacer invisible el arpón de modo que al usuario le parezca que éste ha sido destruido. Es utilizado cuando el arpón colisiona con algún otro objeto o cuando ha alcanzado su altura máxima. Recibe y devuelve void.
- **Disparar()**: éste método se encarga de hacer visible el arpón para que empiece a desplazarse hacia arriba. Recibe y devuelve void.

Clase Bola

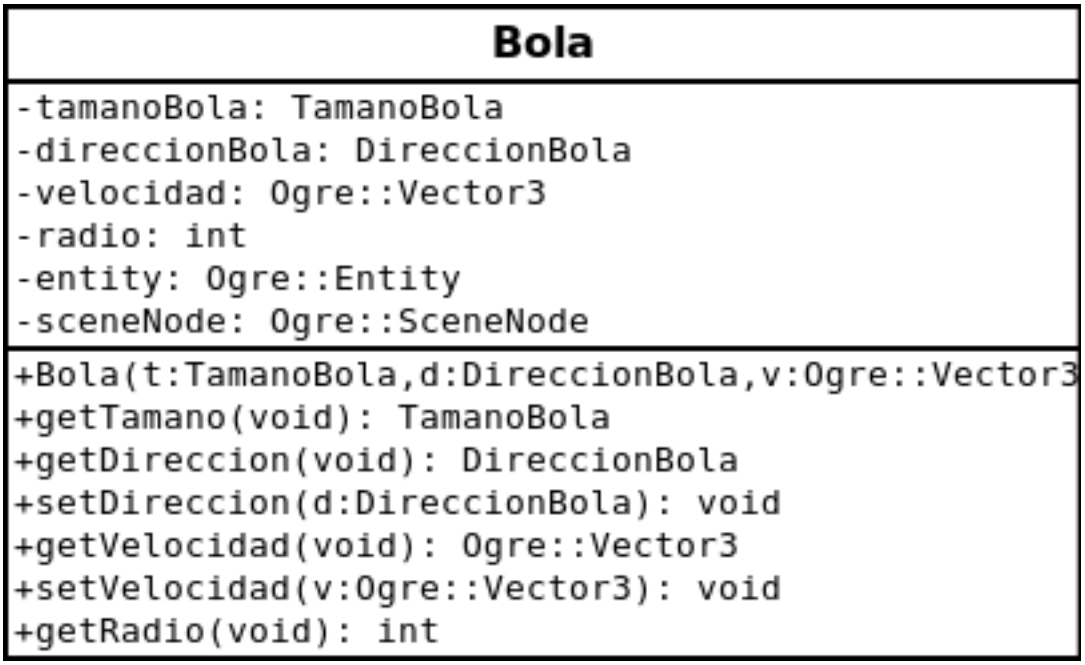

Figura 5.3: Clase Bola

La clase Bola se utiliza para describir las propiedades y el comportamiento de las bolas o globos del juego. Sirve para representar los tres tipos de bolas que aparecen en el juego. Sus atributos y métodos son:

Atributos:

- **tamanoBola**: tamanoBola contiene el tamaño de la bola. Es del tipo enumerado TamanoBola definido también en Bola.h y puede tomar los valores GRANDE, MEDIANA y PEQUENA.
- **direccionBola)**: contiene la dirección a la que se dirige la bola actualmente. Es del tipo enumerado DireccionBola y puede tomar los valores NO, SO, NE y SE haciendo referencia al Noroeste, Sudoeste, Nordeste y Sudeste respectivamente.
- **velocidad**: variable del tipo Ogre::Vector3, es decir, un vector de 3 flotantes. Esta variable se utiliza para controlar la velocidad vertical a la que la bola se mueve.
- **radio**: variable del tipo int que contiene la medida de la bola en formato numérico.
- **entity**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D (modelo, animación, texturas, materiales...).
- **sceneNode**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D entre otras.

Métodos:

- **Bola()**: el constructor de la clase. Recibe 3 parámetros: t del tipo TamanoBola, d del tipo DireccionBola y v del tipo Ogre::Vector3.
- **getTamano()**: método observador que devuelve el contenido del atributo tamanoBola.
- **getDireccion()**: método observador que devuelve el contenido del atributo direccionBola.
- **setDireccion()**: método de escritura que establece el valor del atributo direccionBola. Recibe un parámetro del tipo DireccionBola.
- **getVelocidad()**: método observador que devuelve el contenido del atributo velocidad.
- **setVelocidad()**: método de escritura que establece el valor del atributo velocidad. Recibe un parámetro del tipo Ogre::Vector3.
- **setVelocidad()**: método observador que devuelve el contenido del atributo radio.

Clase Carpintero

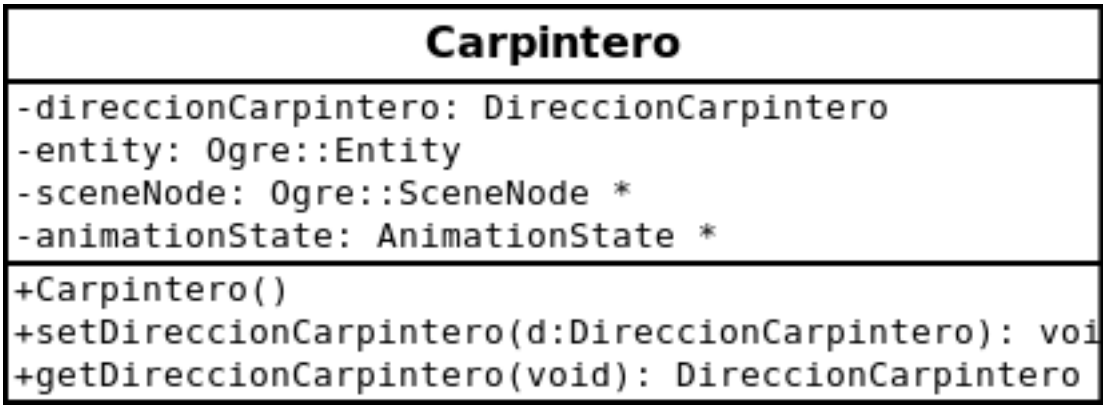

Figura 5.4: Clase Carpintero

La clase Carpintero contiene métodos y atributos que definen el comportamiento de los pájaros carpinteros que aparecen volando en la pantalla destruyendo bolas o al mismo personaje al tocarlos. Contiene los siguientes atributos y métodos:

Atributos:

- **direccionCarpintero**: atributo del tipo enumerado DireccionCarpintero que puede contener los valores N, S, E y O haciendo referencia a los cuatro puntos cardinales.
- **entity**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D (modelo, animación, texturas, materiales...).
- **sceneNode**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D entre otras.
- **animationState**: puntero a un objeto de la clase Ogre::AnimationState que contiene la información importante acerca de la animación del objeto 3D.

Métodos:

- **Carpintero()**: constructor de la clase. No recibe ningún parámetro.
- **setDireccionCarpintero()**: establece el valor para la variable direccionCarpintero. Recibe un parámetro del tipo DireccionCarpintero y devuelve void.
- **getDireccionCarpintero()**: método observador del atributo direccionCarpintero. Recibe void y devuelve un dato del tipo DireccionCarpintero.

Clase Creditos

La clase Creditos contiene métodos y atributos que definen la pantalla de créditos.

Atributos:

- **sceneManager**: atributo del tipo puntero a Ogre::SceneManager, clase que gestiona las escenas de *Ogre*.
- **camera**: atributo del tipo puntero a Ogre::Camera, clase que gestiona la cámara dentro de la escena.
- **overlay**: atributo del tipo puntero a Ogre::Overlay, clase que gestiona las capas con texto y/o imagen que aparecen por encima de la escena.

Métodos:

- **Creditos()**: constructor de la clase.
- **Activar()**: su función es cambiar de cualquier otra escena a la escena de créditos.
- **getOverlay()**: método observador del atributo overlay. Devuelve un puntero a overlay.

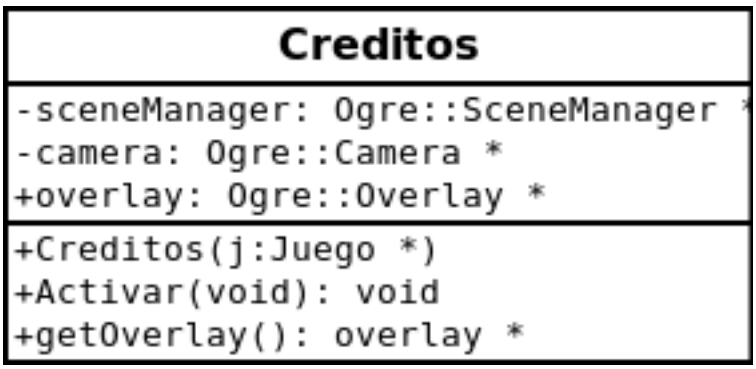

Figura 5.5: Clase Creditos

Clase Cubo

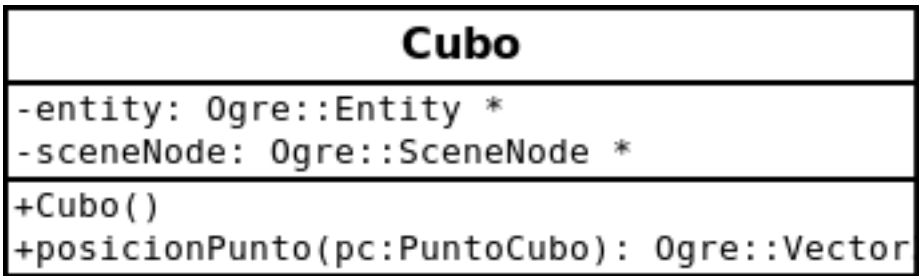

Figura 5.6: Clase Cubo

La clase Cubo contiene métodos y atributos que definen los cubos o plataformas que aparecen durante las fases del juego y pueden ser destruidos con el arpón del personaje.

Atributos:

- **entity**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D (modelo, animación, texturas, materiales...).
- **sceneNode**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D entre otras.

Métodos:

- **Cubo()**: constructor de la clase.
- **PosicionPunto()**: recibe un enumerado del tipo PuntoCubo que puede tomar los valores SSO, SNO, SSE, SNE, ISO, INO, ISE o INE que hacen referencia a los diferentes vértices del cubo (superior sudoeste, superior noroeste...). Devuelve un tipo Ogre::Vector3 con las coordenadas del vértice.

Clase Ermitano

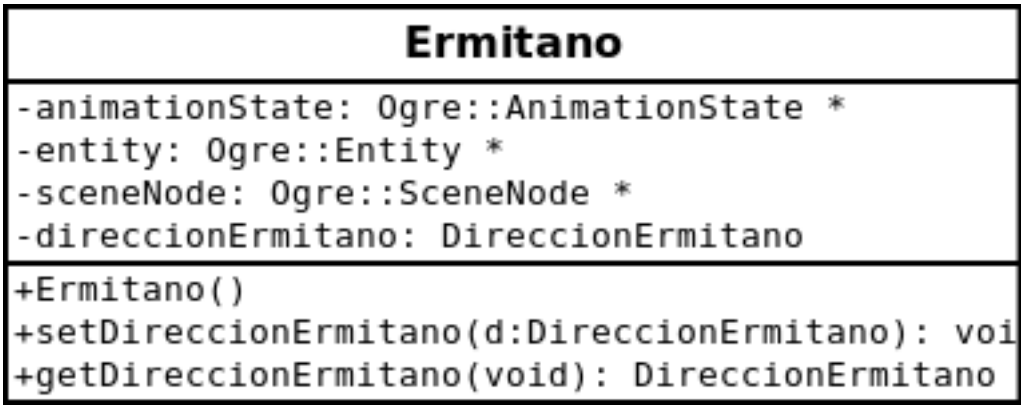

Figura 5.7: Clase Ermitano

La clase Ermitano contiene métodos y atributos que definen los cangrejos ermitaños que aparecen durante las fases del juego y pueden ser destruidos con el arpón del personaje.

Atributos:

- **animationState**: puntero a un objeto de la clase Ogre::AnimationState que contiene la información importante acerca de la animación del objeto 3D.

- **entity**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D (modelo, animación, texturas, materiales...).

- **sceneNode**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D entre otras.

- **direccionErmitano**: atributo del tipo enumerado DireccionErmitano que puede contener los valores N, S, E y O haciendo referencia a los cuatro puntos cardinales.

Métodos:

- **Ermitano()**: constructor de la clase. No recibe ningún parámetro.

- **setDireccionErmitano()**: establece el valor para la variable direccionErmitano. Recibe un parámetro del tipo DireccionErmitano y devuelve void.

- **getDireccionErmitano()**: método observador del atributo direccionErmitano. Recibe void y devuelve un dato del tipo DireccionErmitano.

Clase Fase

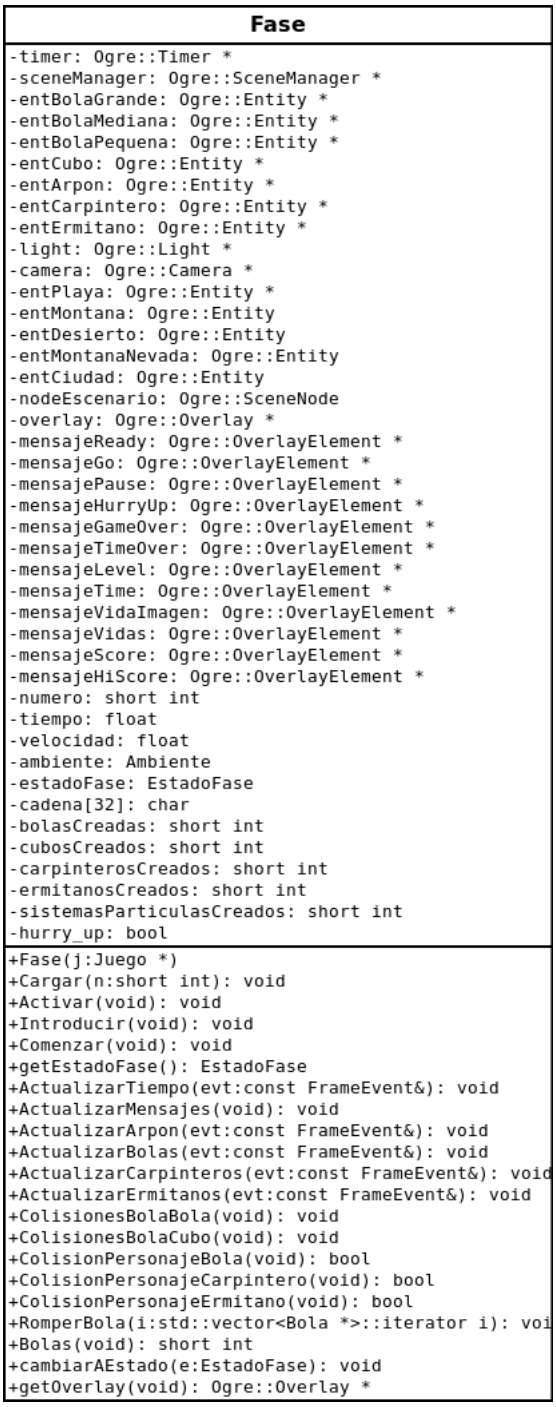

Figura 5.8: Clase Fase

La clase Fase es la más compleja del juego. Contiene métodos y atributos que definen las fases del juego la cual controla al resto de elementos que se encuentran en ella.

Atributos:

- **timer**: puntero a un objeto de la clase Ogre::Timer que contiene información del tiempo. Se utiliza para controlar el tiempo de las fases.
- **sceneManager**: atributo del tipo puntero a Ogre::SceneManager, clase que gestiona las escenas de *Ogre*.
- **entBolaGrande**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D de una bola o globo grande.
- **entBolaMediana**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D de una bola o globo mediano.
- **entBolaPequena**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D de una bola o globo pequeño.
- **entBolaCubo**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D de un cubo o plataforma.
- **entBolaArpon**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D de un arpón.
- **entBolaCarpintero**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D de de un pájaro carpintero.
- **entBolaErmitano**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D de un cangrejo ermitaño
- **light**: atributo del tipo puntero a Ogre::Light, clase que gestiona la luz dentro de la escena.
- **camera**: atributo del tipo puntero a Ogre::Camera, clase que gestiona la cámara dentro de la escena.
- **entPlaya**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D del escenario Playa.
- **entMontana**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D del escenario Montaña.
- **entDesierto**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D del escenario Desierto.
- **entMontanaNevada**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D del escenario Montaña Nevada.
- **entCiudad**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D del escenario Ciudad.
- **nodeEscenario**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D del escenario.
- **overlay**: atributo del tipo puntero a Ogre::Overlay, clase que gestiona las capas con texto y/o imagen que aparecen por encima de la escena.
- **mensajeReady**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje Ready que aparece al inicio del nivel.

- **mensajeGo**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje Go que aparece al inicio del nivel.
- **mensajePause**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje Pause que aparece cuando se pausa la partida.
- **mensajeHurryUp**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje Hurry Up que aparece cuando queda poco tiempo en la fase.
- **mensajeGameOver**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje Game Over que aparece cuando se termina la partida.
- **mensajeTimeOver**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje Time Over que aparece cuando se agota el tiempo de un nivel.
- **mensajeLevel**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje del número de nivel o Level.
- **mensajeTime**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje del tiempo restante.
- **mensajeVidaImagen**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso la imagen que aparece junto a las vidas del personaje.
- **mensajeVidas**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje con las vidas que le queda al personaje.
- **mensajeScore**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje con la puntuación actual o Score.
- **mensajeHiScore**: atributo del tipo puntero a Ogre::OverlayElement, que contiene un elemento de una capa u overlay, en este caso el mensaje con la puntuación más alta o Hi-Score.
- **numero**: atributo del tipo short int. Contiene el número de la fase que se está jugando.
- **tiempo**: atributo del tipo float. Contiene el tiempo actual que le queda al jugador para pasar de fase.
- **velocidad**: atributo del tipo float. Contiene un valor que indica la velocidad a la que se mueven las bolas o globos y los animales de las fases. Cuanto mayor sea su valor mayor velocidad habrá.
- **ambiente**: atributo del tipo enumerado Ambiente que puede tomar los valores DIA, TARDE o NOCHE. Dependiendo del valor que tome se mostrará la luz en una posición y de un color diferente.
- **estadoFase**: atributo del tipo enumerado EstadoFase que puede tomar los valores INTRODUCCION, PERDIENDO, JUGANDO, VENCIENDO, GAME_OVER, TIME_OVER o PAUSA. Dependiendo del estado de la fase la clase OyenteFase actúa de forma diferente.
- **cadena**: cadena de caracteres de bajo nivel de longitud 32. Se utiliza para escribir ciertos mensajes en pantalla.

- **bolasCreadas**: atributo del tipo short int. Contiene el número de bolas o globos que se han creado para ser nombrados al crearlos.
- **cubosCreados**: atributo del tipo short int. Contiene el número de cubos creados para ir nombrándolos a medida que son creados.
- **ermitanosCreados**: atributo del tipo short int. Contiene el número de cangrejos ermitaños creados para ir nombrándolos a medida que son creados.
- **carpinterosCreados**: atributo del tipo short int. Contiene el número de pájaros carpinteros creados para ir nombrándolos a medida que son creados.
- **sistemasParticulasCreados**: atributo del tipo short int. Contiene el número de sistemas de partículas creados para ir nombrándolos a medida que son creados.
- **hurryUp**: atributo del tipo bool. Indica si se ha mostrado o no el mensaje "Hurry Up!".

Métodos:

- **Fase()**: constructor de la clase.
- **Cargar()**: recibe un parámetro del tipo short int. Carga la fase desde un fichero XML cuyo nombre contiene el número pasado como parámetro.
- **Activar()**: su función es cambiar de cualquier otra escena a la escena de créditos.
- **Introducir()**: muestra la introducción de la fase que se realiza al comenzarse. Recibe y devuelve void.
- **Comenzar()**: comienza el nivel. Recibe y devuelve void.
- **getEstadoFase()**: método observador del atributo estadoFase. Devuelve un tipo enumerado del tipo EstadoFase.
- **ActualizarTiempo()**: recibe un parámetro del tipo Ogre::Evt. Actualiza el tiempo restante.
- **ActualizarMensajes()**: actualiza los mensaje de la capa superior u overlay. Recibe y devuelve void.
- **ActualizarArpon()**: actualiza los mensaje de la capa superior u overlay. Recibe y devuelve void.
- **ActualizarBolas()**: actualiza los mensaje de la capa superior u overlay. Recibe y devuelve void.
- **ActualizarCarpinteros()**: recibe un parámetro del tipo Ogre::Evt. Actualiza la posición y animación de los pájaros carpinteros que aparecen en las fases.
- **ActualizarErmitanos()**: recibe un parámetro del tipo Ogre::Evt. Actualiza la posición y animación de los cangrejos ermitaños que aparecen en las fases.
- **ColisionesBolaBola()**: recibe y devuelve void. Detecta y gestiona las colisiones entre las bolas o globos. Tiene un grado de complejidad n^2 ya que por cada objeto del tipo Bola (n objetos) hay que realizar n-1 comparaciones.
- **ColisionesBolaCubo()**: recibe y devuelve void. Detecta y gestiona las colisiones entre las bolas o globos y Cubos. Tiene un grado de complejidad n^2 ya que por cada objeto del tipo Bola (n objetos) hay que realizar m (número de cubos) comparaciones.

- **ColisionPersonajeBola()**: recibe void y devuelve bool. Dicho valor devuelto tomará el valor true si se detecta una colisión entre el personaje y alguna bola.
- **ColisionPersonajeCarpintero()**: recibe void y devuelve bool. Dicho valor devuelto tomará el valor true si se detecta una colisión entre el personaje y algún pájaro carpintero.
- **ColisionPersonajeErmitano()**: recibe void y devuelve bool. Dicho valor devuelto tomará el valor true si se detecta una colisión entre el personaje y algún cangrejo ermitaño.
- **RomperBola()**: recibe un iterador a un vector de objetos del tipo Bola. Devuelve void. Se encarga de hacer desaparecer una bola y crear cuatro nuevas de menor tamaño si la bola destruida no es pequeña.
- **Bolas()**: recibe void y devuelve un valor del tipo short int que indica el numero de objetos tipo Bola que existen.
- **cambiarAEstado()**: recibe un parámetro del tipo EstadoJuego y devuelve void. Hace que el objeto fase cambie de estado realizando las operaciones oportunas.
- **getOverlay()**: método observador del atributo overlay. Recibe void y devuelve un puntero a Ogre::Overlay.

Clase Juego

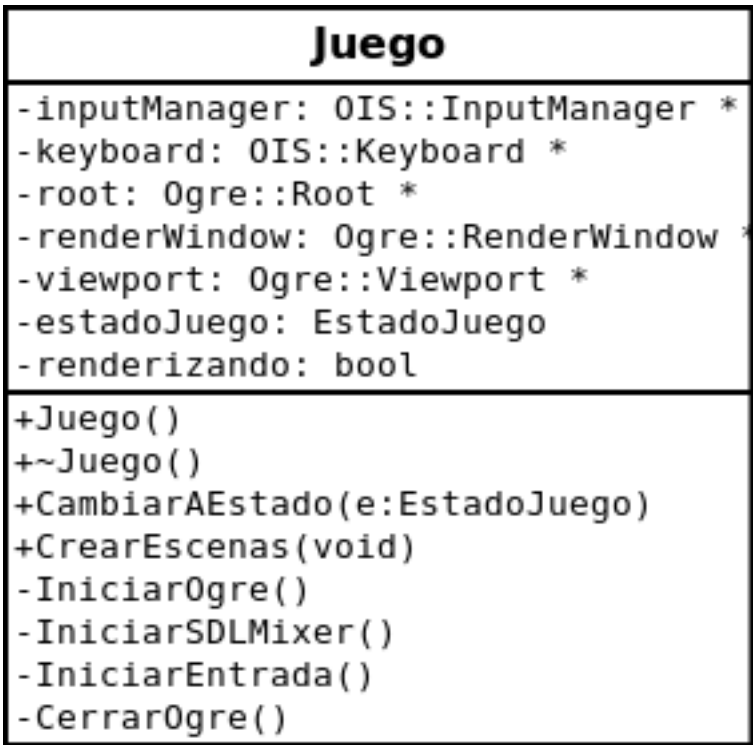

Figura 5.9: Clase Juego

La clase Juego contiene los atributos y métodos necesarios para la gestión general de la aplicación, es decir, inicializarla, cerrarla, cargar los objetos principales, etcétera...

Atributos:

- **inputManager**: atributo del tipo puntero a OIS::InputManager que gestiona la entrada, en el caso de *Balloon Breakers* la entrada es el teclado.
- **keyboard**: atributo del tipo puntero a OIS::Keyboard. La clase que controla el teclado.
- **root**: atributo del tipo puntero a Ogre::Root. Root es la clase raíz de *Ogre* desde la cual se accede al resto de clases.
- **renderWindow**: atributo del tipo puntero a Ogre::RenderWindow, la ventana donde se renderizan los objetos 3D, luces, etcétera.
- **viewport**: atributo del tipo puntero a Ogre::Viewport, un marco donde se visualiza la escena 3D.
- **estadoJuego**: atributo del tipo puntero a EstadoJuego, un enumerado que puede tomar los valores INICIO, MENUPRINCIPAL, PARTIDA o CREDITOS dependiendo del estado en que se encuentre la aplicación.
- **renderizando**: atributo del tipo bool. Su valor cambia a true cuando se ha comenzado el bucle de renderizado de *Ogre*.

Métodos:

- **Juego()**: constructor de la clase. No recibe ningún parámetro.
- **Juego()**: destructor de la clase. Llama a los destructores de los objetos que han sido creados durante el juego.
- **CambiarAEstado()**: recibe un parámetro del tipo puntero a EstadoJuego y establece el valor de la variable estadoJuego.
- **CrearEscenas()**: recibe void. Crea las escenas básicas del juego: la del menú principal, los créditos y las fases.
- **IniciarOgre()**: no recibe ni devuelve nada. Se encarga de inicializar el motor de renderizado.
- **IniciarSDLMixer()**: no recibe ni devuelve nada. Se encarga de inicializar el gestor de sonido y música.
- **IniciarEntrada()**: no recibe ni devuelve nada. Se encarga de inicializar el gestor de la entrada.
- **CerrarOgre()**: no recibe ni devuelve nada. Se encarga de cerrar el motor de renderizado.

Clase MenuPrincipal

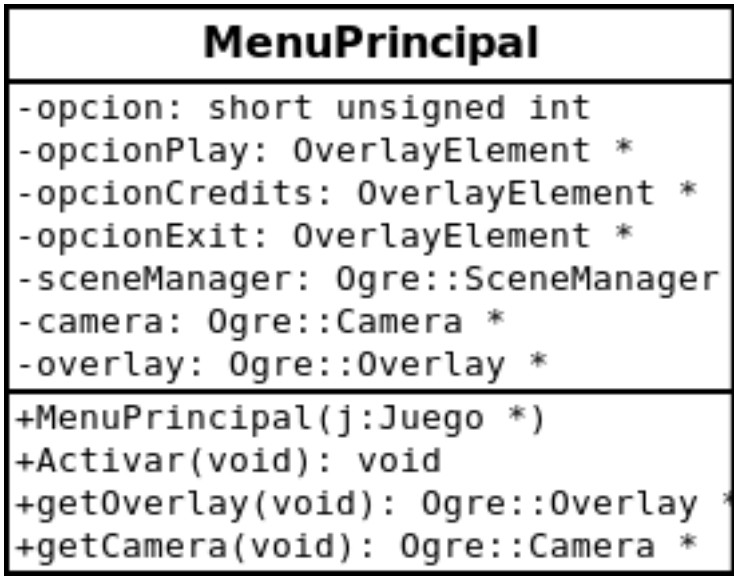

Figura 5.10: Clase MenuPrincipal

La clase MenuPrincipal contiene métodos y atributos que definen la pantalla del menú principal.

Atributos:

- **opcion**: atributo del tipo short unsigned int que contiene la opción seleccionada: 1 para la opción "Play", 2 para "Credits" y 3 para "Exit".
- **opcionPlay**: atributo del tipo puntero a Ogre::OverlayElement y apunta al elemento Play de la capa de opciones. para "Credits" y 3 para "Exit".
- **opcionCredits**: atributo del tipo puntero a Ogre::OverlayElement y apunta al elemento Credits de la capa de opciones. para "Credits" y 3 para "Exit".
- **opcionExit**: atributo del tipo puntero a Ogre::OverlayElement y apunta al elemento Exit de la capa de opciones.
- **camera**: atributo del tipo puntero a Ogre::Camera, clase que gestiona la cámara dentro de la escena.
- **overlay**: atributo del tipo puntero a Ogre::Overlay, clase que gestiona las capas con texto y/o imagen que aparecen por encima de la escena.

Métodos:

- **MenuPrincipal()**: constructor de la clase.
- **Activar()**: su función es cambiar de cualquier otra escena a la escena del menú principal.
- **getOverlay()**: método observador del atributo overlay. Devuelve un puntero a overlay.
- **getCamera()**: método observador del atributo camera. Devuelve un puntero a camera.

Clase OyenteCreditos

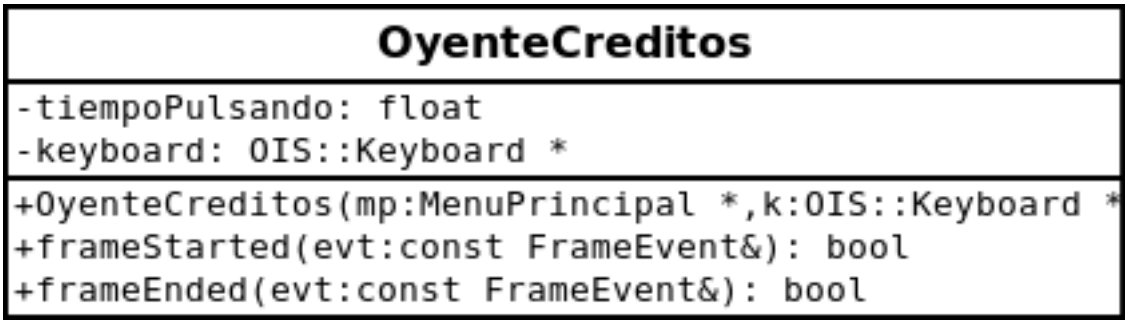

Figura 5.11: Clase OyenteCreditos

La clase OyenteCreditos contiene métodos y atributos que interactúan con la clase Creditos y el teclado. Esta clase es una clase heredada de Ogre::Frame listener. Contiene los siguientes atributos y métodos:

Atributos:

- **tiempoPulsando**: atributo del tipo float que contiene el tiempo que se ha ido pulsando una tecla.
- **keyboard**: puntero a la clase OIS::Keyboard.

Métodos:

- **OyenteCreditos()**: constructor de la clase.
- **frameStarted()**: método que se ejecuta antes de que se dibuje la escena. Recibe un parámetro del tipo Ogre::Event y devuelve un dato del tipo bool que si es true hará que se pare el bucle de renderizado.
- **frameEnded()**: método que se ejecuta antes de que se dibuje la escena. Recibe un parámetro del tipo Ogre::Event y devuelve un dato del tipo bool que si es true hará que se pare el bucle de renderizado.

Clase OyenteFase

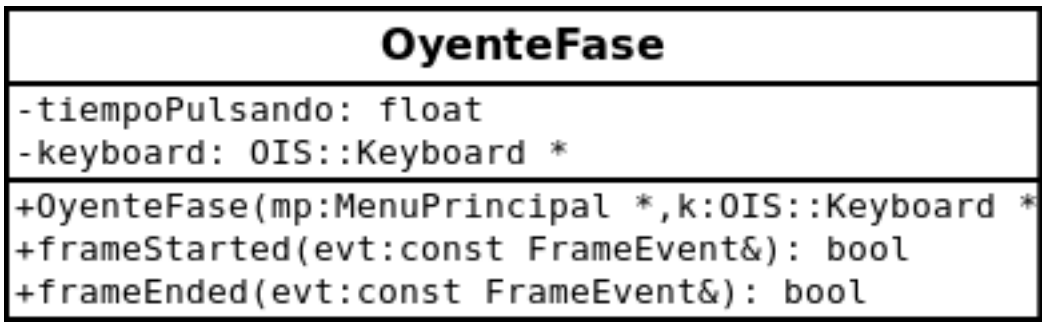

Figura 5.12: Clase OyenteFase

La clase OyenteFase contiene métodos y atributos que interactúan con la clase Fase y el teclado. Esta clase es una clase heredada de Ogre::Frame listener. Contiene los siguientes atributos y métodos:

Atributos:

- **tiempoPulsando**: atributo del tipo float que contiene el tiempo que se ha ido pulsando una tecla.
- **keyboard**: puntero a la clase OIS::Keyboard.

Métodos:

- **OyenteFase()**: constructor de la clase.
- **frameStarted()**: método que se ejecuta antes de que se dibuje la escena. Recibe un parámetro del tipo Ogre::Event y devuelve un dato del tipo bool que si es true hará que se pare el bucle de renderizado.
- **frameEnded()**: método que se ejecuta antes de que se dibuje la escena. Recibe un parámetro del tipo Ogre::Event y devuelve un dato del tipo bool que si es true hará que se pare el bucle de renderizado.

Clase OyenteMenuPrincipal

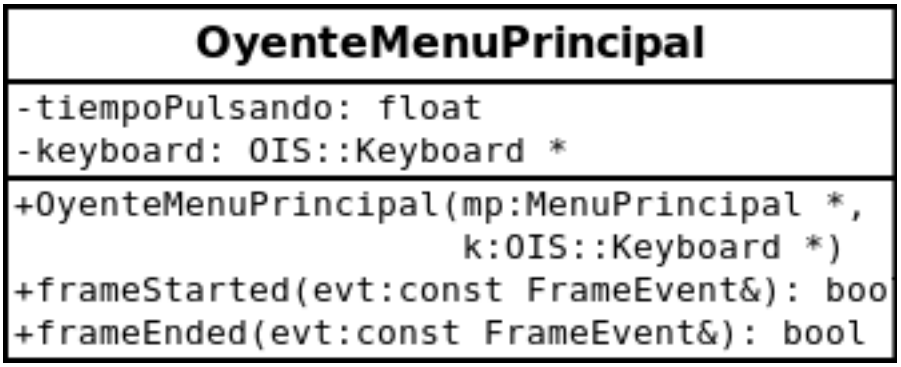

Figura 5.13: Clase OyenteMenuPrincipal

La clase OyenteMenuPrincipal contiene métodos y atributos que interactúan con la clase MenuPrincipal y el teclado. Esta clase es una clase heredada de Ogre::Frame listener. Contiene los siguientes atributos y métodos:

Atributos:

- **tiempoPulsando**: atributo del tipo float que contiene el tiempo que se ha ido pulsando una tecla.
- **keyboard**: puntero a la clase OIS::Keyboard.

Métodos:

- **OyenteMenuPrincipal()**: constructor de la clase.
- **frameStarted()**: método que se ejecuta antes de que se dibuje la escena. Recibe un parámetro del tipo Ogre::Event y devuelve un dato del tipo bool que si es true hará que se pare el bucle de renderizado.
- **frameEnded()**: método que se ejecuta antes de que se dibuje la escena. Recibe un parámetro del tipo Ogre::Event y devuelve un dato del tipo bool que si es true hará que se pare el bucle de renderizado.

Clase Partida

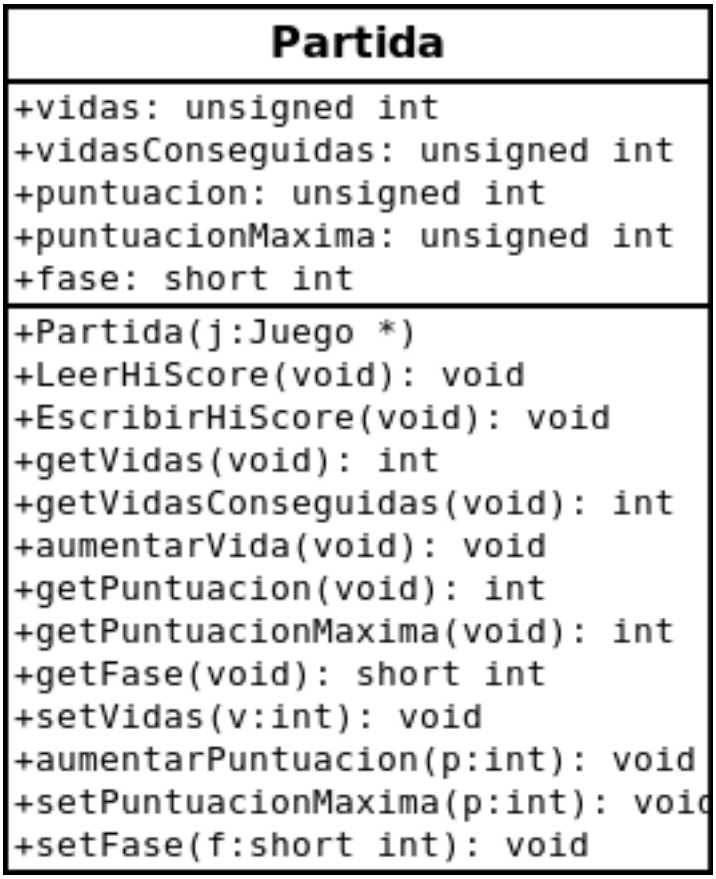

Figura 5.14: Clase Partida

La clase Partida contiene métodos y atributos que gestionan los datos relevantes durante el desarrollo de una partida del juego.

Atributos:

- **vidas**: atributo del tipo unsigned int que contiene las vidas que quedan en una partida.
- **vidasConseguidas**: atributo del tipo unsigned int que contiene las vidas que se han conseguido en una partida.

- **puntuacion**: atributo del tipo unsigned int que contiene la puntuación actual en una partida.
- **puntuacionMaxima**: atributo del tipo unsigned int que contiene la puntuación máxima obtenida.
- **fase**: atributo del tipo short int que contiene el número de fase que se está jugando actualmente.

Métodos:

- **Partida()**: constructor de la clase.
- **LeerHiScore()**: recibe y devuelve void. Lee el archivo de texto con la puntuación máxima y lo guarda en memoria.
- **EscribirHiScore()**: recibe y devuelve void. Escribe el archivo de texto con la puntuación máxima.
- **getVidas()**: método observador del atributo vidas.
- **getVidasConseguidas()**: método observador del atributo vidasConseguidas.
- **aumentarVida()**: recibe y devuelve void. Aumenta en uno el valor de los atributos vidas y vidas-Conseguidas.
- **getPuntuacion()**: método observador del atributo fase.
- **getPuntuacionMaxima()**: método observador del atributo puntuacionMaxima.
- **getFase()**: método observador del atributo fase.
- **setVidas()**: recibe un parámetro del tipo int y devuelve void. Establece el valor del atributo vidas.
- **aumentarPuntuacion()**: recibe un parámetro del tipo int y devuelve void. Aumenta el valor del atributo puntuacion.
- **setPuntuacionMaxima()**: recibe un parámetro del tipo int y devuelve void. Establece el valor del atributo puntuacionMaxima.
- **setFase()**: recibe un parámetro del tipo short int y devuelve void. Establece el valor del atributo fase.

Clase Personaje

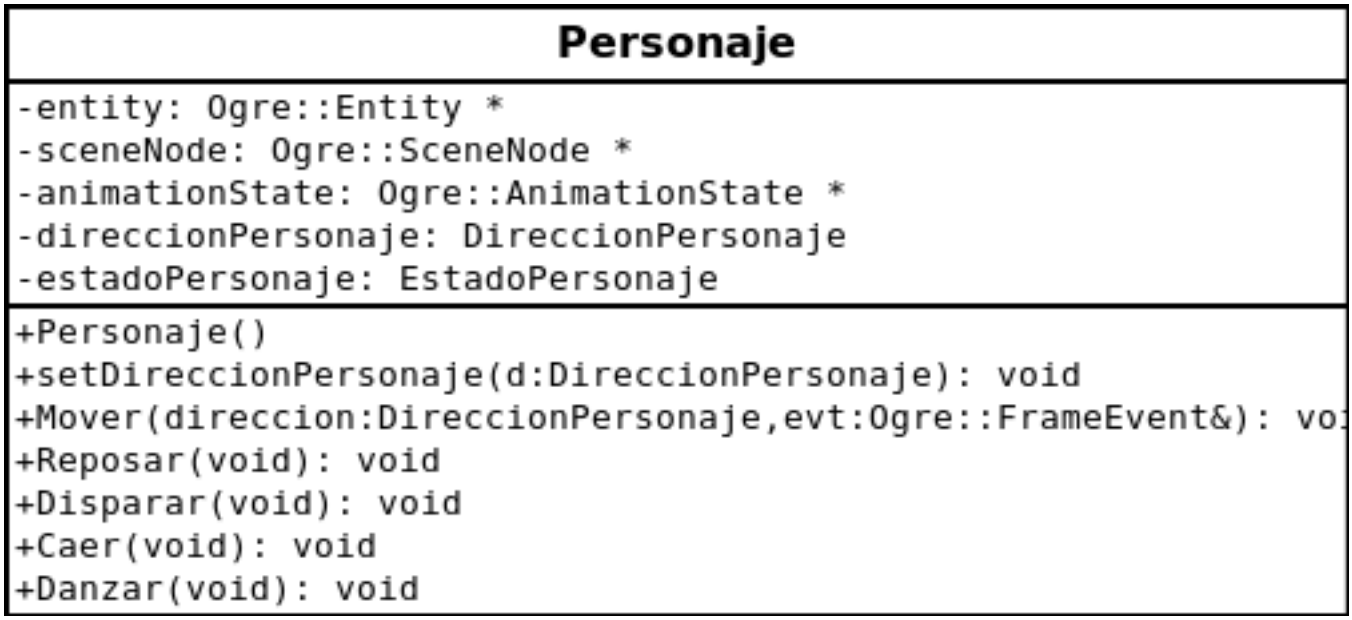

Figura 5.15: Clase Personaje

La clase Personaje contiene métodos y atributos que gestionan los datos relevantes del personaje del juego. Contiene los siguientes atributos y métodos:

Atributos:

- **entity**: puntero a un objeto de la clase Ogre::Entity que almacena el modelo 3D (modelo, animación, texturas, materiales...).
- **sceneNode**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D entre otras.
- **animationState**: puntero a un objeto de la clase Ogre::AnimationState que contiene la información importante acerca de la animación del objeto 3D.
- **direccionErmitano**: atributo del tipo enumerado DireccionPersonaje que puede contener los valores N, S, E y O haciendo referencia a los cuatro puntos cardinales.
- **estadoPersonaje**: atributo del tipo enumerado EstadoPersonaje que puede tomar los valores REPOSO, CORRIENDO, DISPARANDO, CAYENDO o DANZANDO.

Métodos:

- **Personaje**(): constructor de la clase.
- **setDireccionPersonaje**(): constructor de la clase.

Clase SistemaParticulas

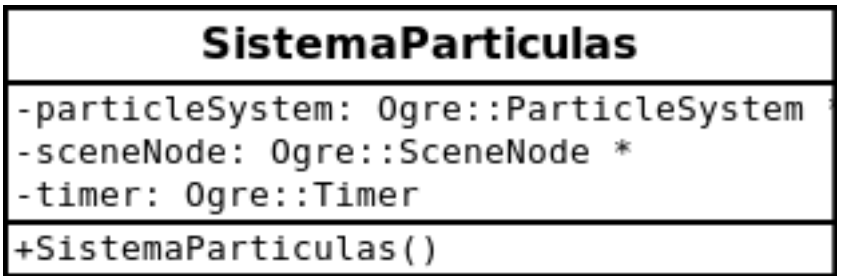

Figura 5.16: Clase SistemaParticulas

La clase SistemaParticulas contiene métodos y atributos que gestionan efectos de partículas que aparecen durante las fases al romper un globo o al destruir un animal. Contiene los siguientes atributos y métodos:

Atributos:

- **particleSystem**: atributo del tipo puntero a Ogre::ParticleSystem.
- **sceneNode**: puntero a un objeto de la clase Ogre::SceneNode que contiene información de la posición y dirección del objeto 3D entre otras.
- **timer**: puntero a un objeto de la clase Ogre::Timer que contiene información temporal.

Métodos:

- **SistemaParticulas**(): constructor de la clase.

Clase Sonido

Figura 5.17: Clase Sonido

La clase Sonido contiene métodos y atributos para la gestión, a través de *SDL_mixer*, del sonido y la música del juego. Contiene los siguientes atributos y métodos:

Atributos:

- **cancion**: puntero a una estructura Mix_Music de *SDL_mixer* para la gestión de la canción que suena actualmente.

- **sonidos**: vector de bajo nivel de punteros a estructuras Mix_Chunk para la gestión de sonidos.

Métodos:

- **Sonido**(): constructor de la clase.

- **CargarSonidos**(): recibe y devuelve void. Se encarga de cargar los sonidos en el vector sonidos.

- **CargarCancion**(): recibe una cadena de bajo nivel (const char *) y devuelve void. Se encarga de cargar una canción en memoria a la que apunta el puntero cancion.

5.2. Diseño de las fases

Durante el diseño de *Balloon Breakers* ha sido importante tener una idea clara de cómo el juego iba a ser.

Las fases del juego, dónde se desarrolla toda la acción de éste, tenían que tener unas medidas en el mundo 3D adecuadas para que el juego fuera divertido y jugable.

5.2.1. Medidas de los elementos

Tuve una idea de cómo los escenarios de juego tenían que ser y plasmé mi idea en un primer boceto.

Al ser el entorno de *Ogre* y *Blender* un entorno virtual las medidas no vienen expresadas en una unidad conocida ya que lo importante es la relación entre dichas medidas.

A continuación muestro el resultado del boceto con las medidas de los elementos que conforman las fases:

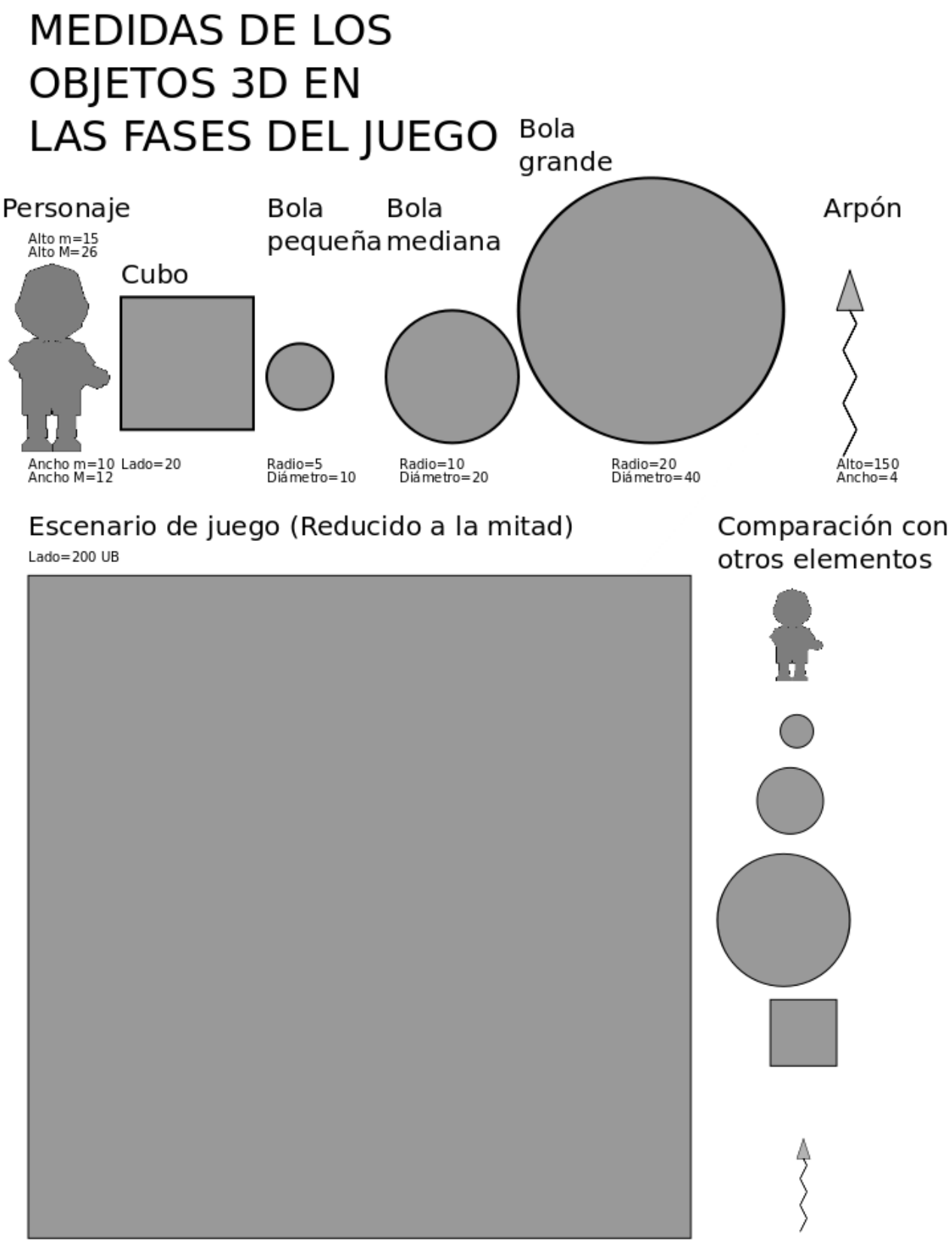

Figura 5.18: Medidas relativas de los elementos de las fases

5.2.2. Estructura de los ficheros XML de las fases

Los ficheros XML que contienen la descripción de cada fase son ficheros XML.

Estos ficheros tienen la siguiente estructura:

```
<?xml version="1.0" standalone=no>

<Fase numero="1" cancion="nombre.extension" tiempo="150" velocidad="1"
escenario="playa" ambiente="tarde">
  <Objeto tipo="bola" tamano="mediana"
  direccion="SO" px="0" py="110" pz="0" />
  <Objeto tipo="cubo" px="-40"
  py="80" pz="-20" />
  <Objeto tipo="carpintero" direccion="O"
  px="800" py="40" pz="0" />
  <Objeto tipo="ermitano" direccion="S"
  px="60" py="5" pz="0" />
</Fase>
```

La primera línea hace referencia a la versión de XML que se está usando.

Posteriormente se define la fase con la etiqueta Fase donde se indica el número, la canción, el tiempo, la velocidad, el escenario y el ambiente en el que se jugará.

Dentro de la etiqueta Fase se definen los objetos de la fase con etiquetas Objeto. Estas etiquetas contienen el tipo que puede ser bola, cubo, carpintero o ermitano.

Si el objeto es del tipo bola la etiqueta tamano definirá el tamaño de esta que puede ser grande, mediana o pequena. La dirección se establece mediante la etiqueta direccion y puede tener los valores SO, NO, SE o NE haciendo referencia a las cuatro direcciones en las que las bolas se mueven. Por último los atributos px, py y pz establecen la posición donde se encuentran las bolas.

Si el objeto es del tipo cubo este sólo tendrá los atributos px, py y pz que definen la posición del cubo.

Si el objeto es del tipo carpintero o ermitano (ermitaño), este tendrá el atributo direccion, que puede tomar los valores N, S, E y O haciendo referencia a los cuatro puntos cardinales y los atributos px, py y pz como se ha definido anteriormente.

5.3. Diseño multimedia

Para el diseño multimedia tuve varias opciones: buscar archivos libres ya existentes o crear los míos.

Finalmente, diseñé personalmente todos el apartado gráfico y utilicé sonidos, voces y música existente que posteriormente modifiqué con *Audacity*. Todo ello se explica en los siguientes apartados.

5.3.1. Diseño gráfico

Para el diseño gráfico me decidí por la primera opción he hice el diseño 3D, 2D yo mismo utilizando *Blender* y *Gimp*.

Los archivos de Blender son fácilmente exportables al formato .mesh con el que *Ogre* trabaja.

Modelos 3D

Quería buscar un aspecto alegre, con cierto aspecto a dibujos animados o comics de estética oriental o *Manga*.

Diseñé cada modelo 3D agregándole una textura creada en *Gimp*. Las animaciones de los modelos Personaje, Ermitaño y Carpintero se hicieron también en *Blender*.

En total creé los siguientes modelos 3D para *Balloon Breakers*:

- **Personaje**: Un personaje sencillo con un pañuelo en la cabeza y ropas de camuflaje, armado con un arpón y calzado con unos zapatos con el dibujo de Supertux.

Figura 5.19: Personaje de frente

- **Arpón**: Un sencillo arpón en espiral.

Figura 5.20: Arpón

- **Carpintero**: Un pájaro carpintero de colores rojo, gris y negro.

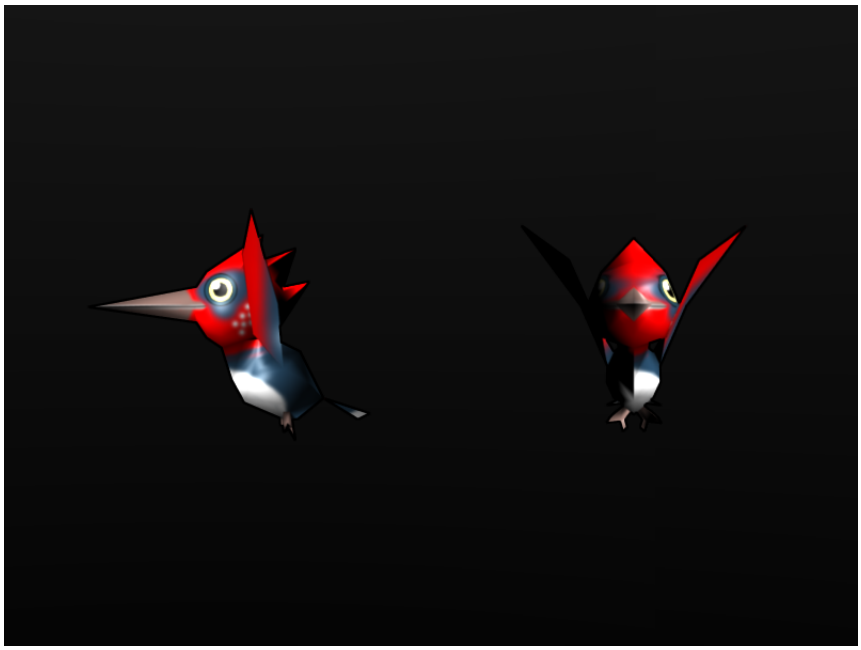

Figura 5.21: Pájaro carpintero

- **Ermitaño**: Un cangrejo ermitaño cuyas pinzas y caparazón pueden pinchar las bolas.

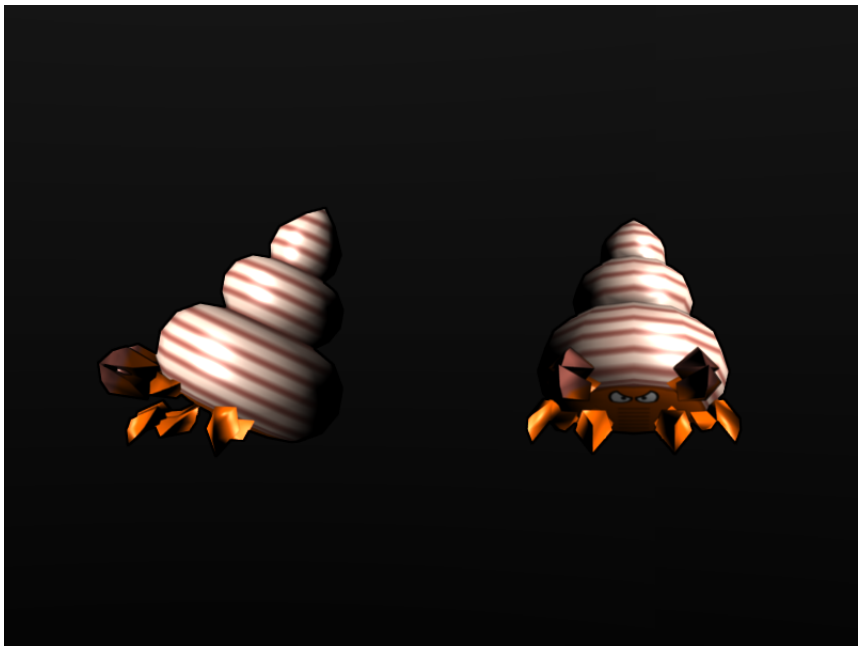

Figura 5.22: Cangrejo ermitaño

- **Bolas**: Tres bolas de diferente tamaño y color.

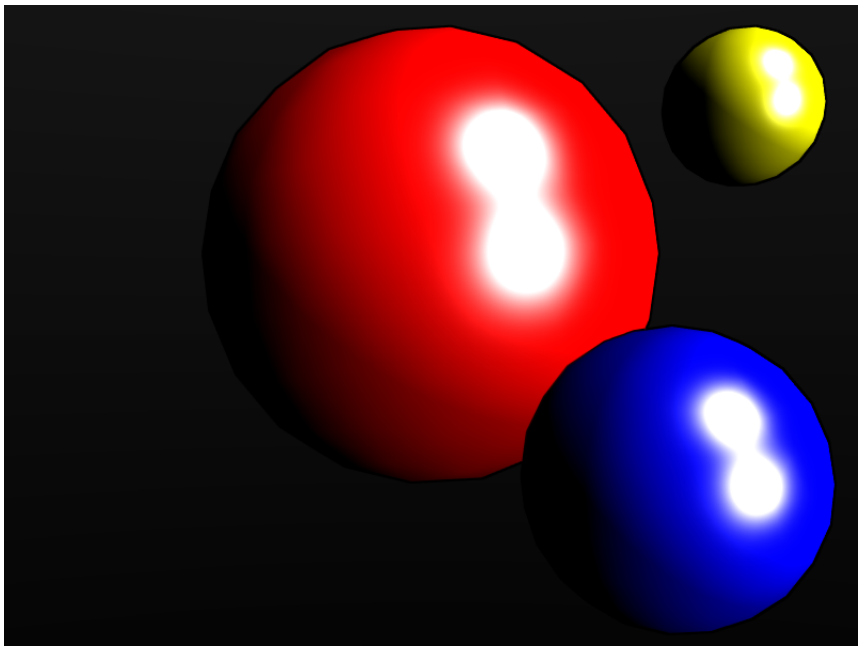

Figura 5.23: Bolas o globos

- **Cubos**: Cubos con el logo del juego que pueden ser destruidos para acceder a las bolas superiores.

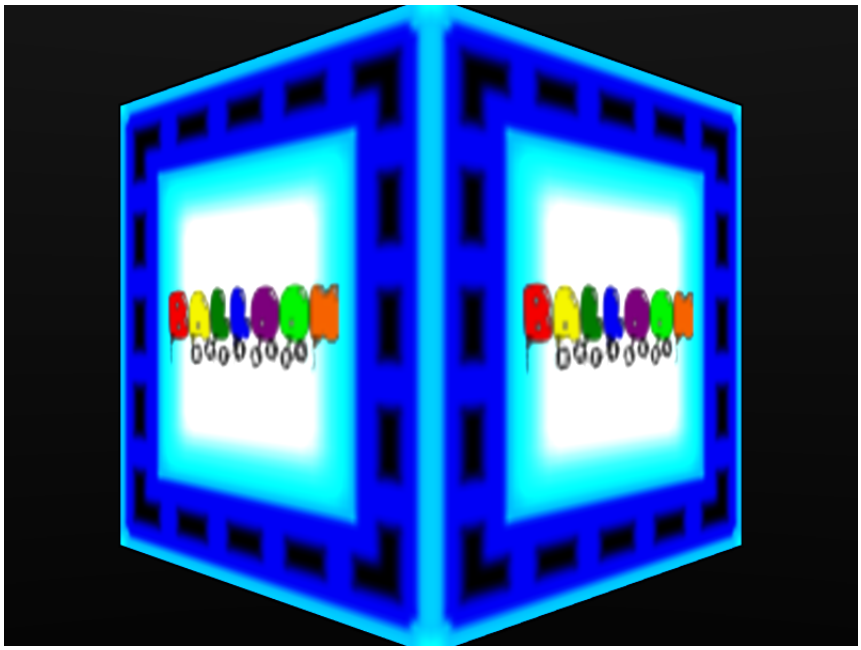

Figura 5.24: Cubos

- **Playa**: Una paradisiaca playa desierta con palmeras y arbustos.

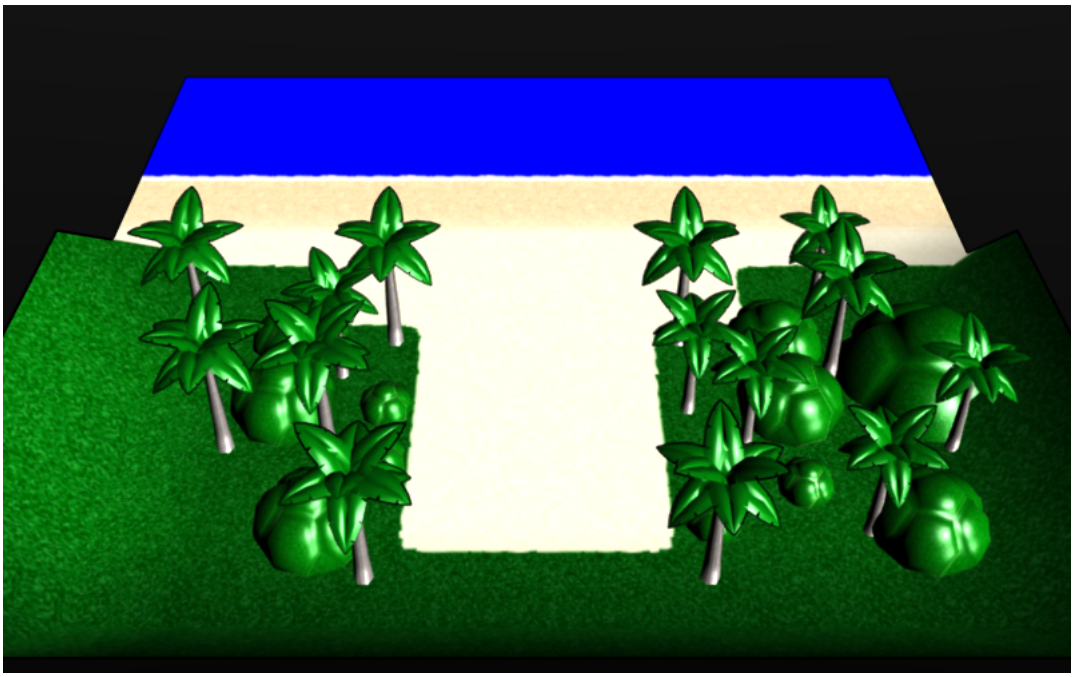

Figura 5.25: Playa

- **Montaña**: Una montaña con un acantilado y un prado verde con bonitos pinos.

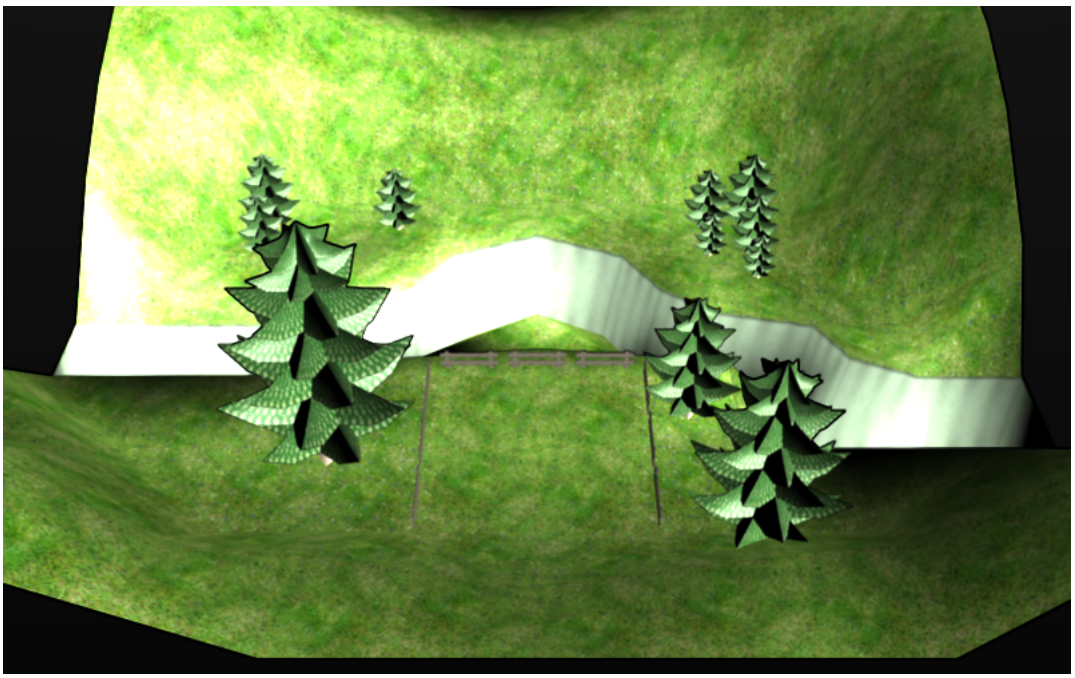

Figura 5.26: Montaña

- **Desierto**: Un desierto al estilo Cañón del Colorado, con cactus, plantas desérticas.

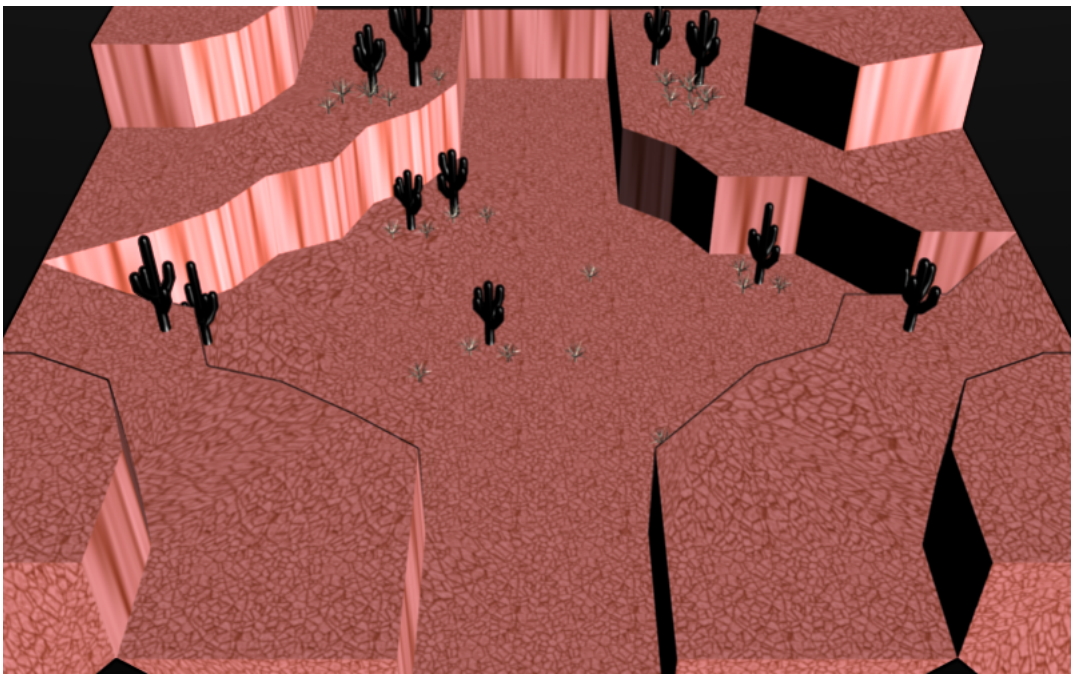

Figura 5.27: Desierto

- **Montaña nevada**: La misma montaña con acantilado pero cubierta de nieve.

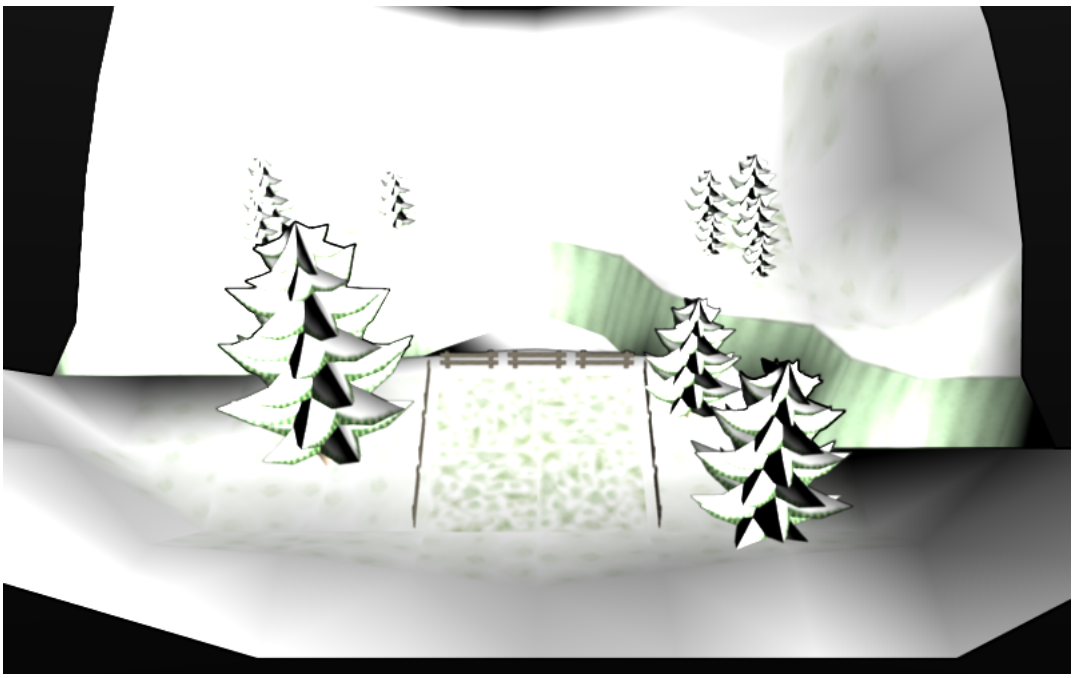

Figura 5.28: Montaña nevada

- **Ciudad**: Una ciudad fantasma con altos edificios.

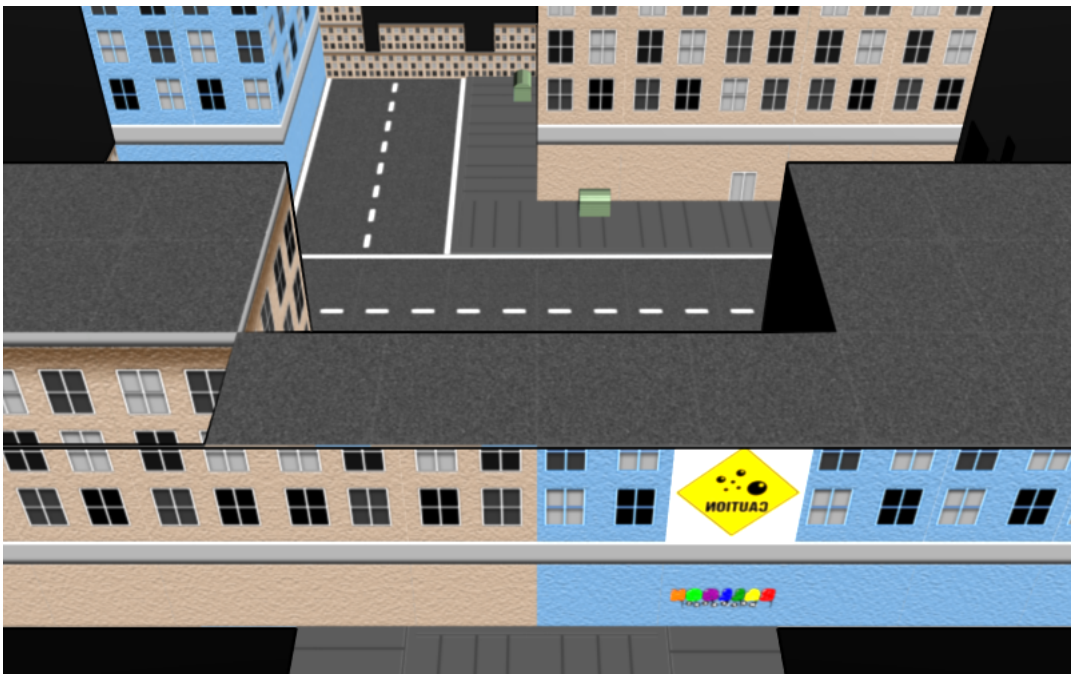

Figura 5.29: Ciudad

Sky Boxes

Los Sky Boxes o Cajas de Cielo son cubos convexos que representan el cielo en el mundo 3D. Para *Balloon Breakers* creé tres diferentes cielos o atmósferas: día, tarde y noche. Bastó con crear una imagen cuadrada para cada cara de cada cubo y asignarle, mediante código, dicha imagen a cada cara. A continuación se muestran, aplanadas, las tres atmósferas creadas para el día, la tarde y la noche del juego:

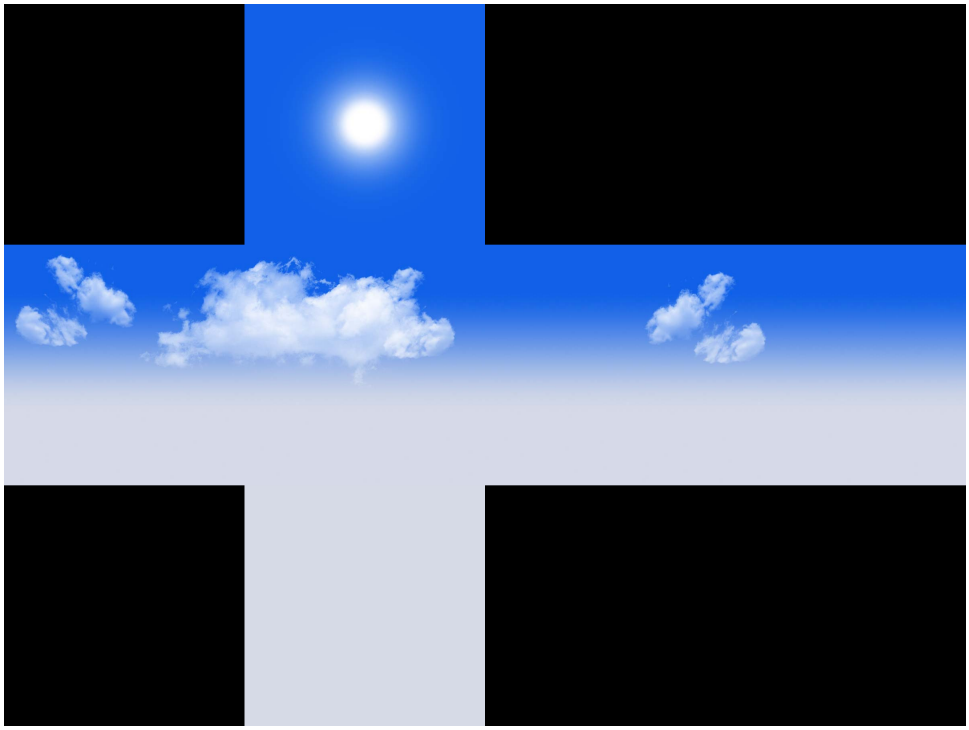

Figura 5.30: Texturas del Sky Box para representar el día

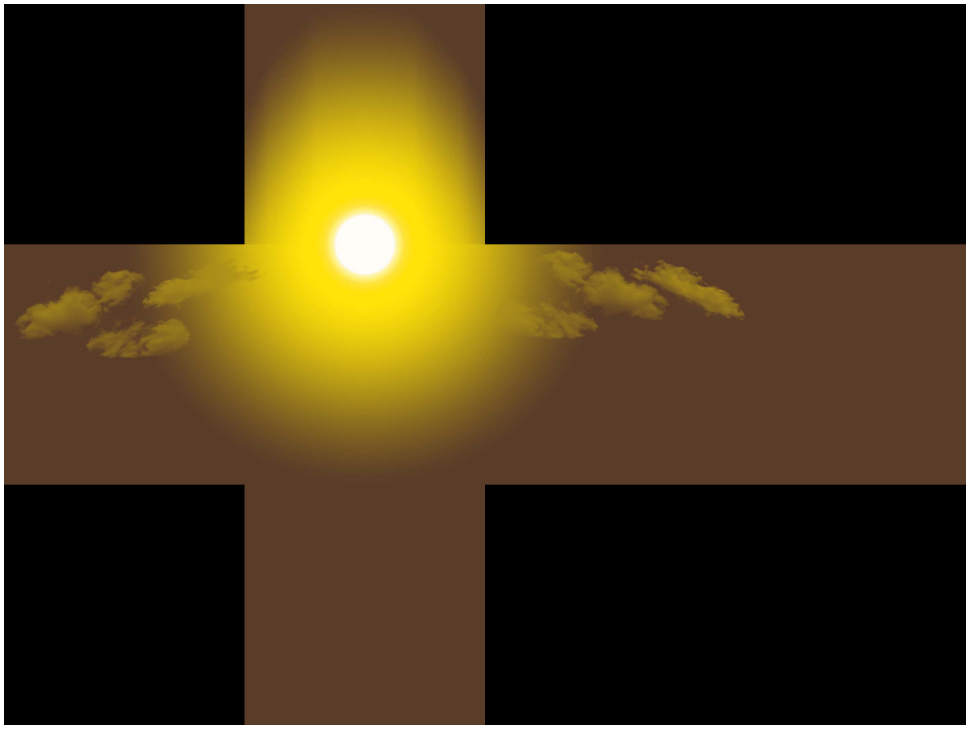

Figura 5.31: Texturas del Sky Box para representar la tarde

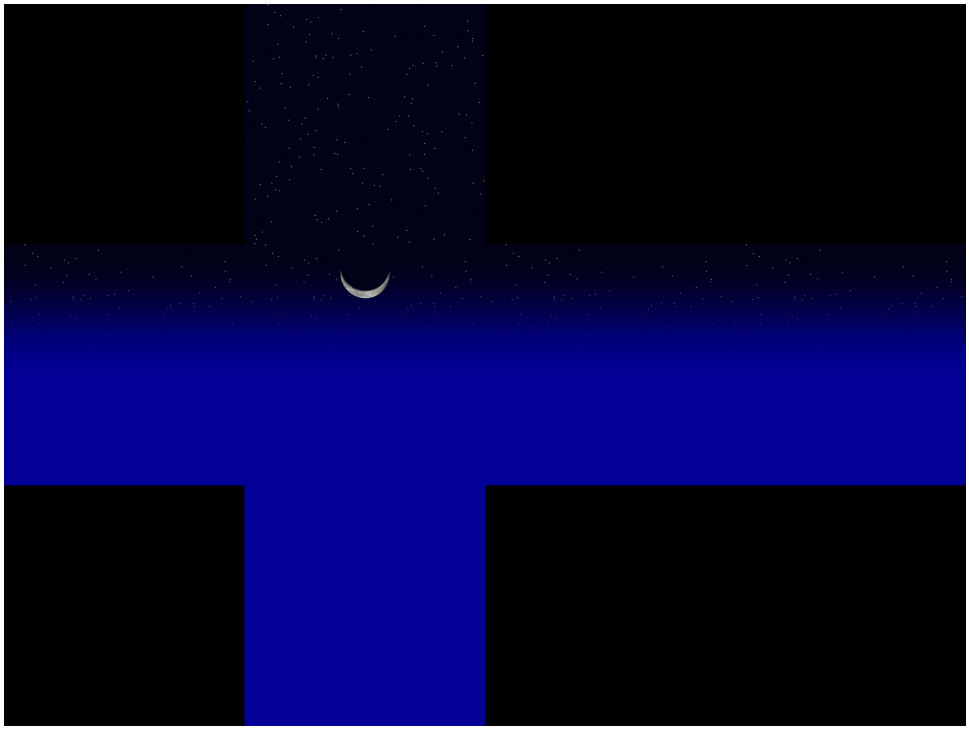

Figura 5.32: Texturas del Sky Box para representar la noche

Capas

Para el diseño de los menús realicé diferentes archivos .overlay fácilmente configurables y con los que *Ogre* trabaja directamente.

Algunos de estos elementos incluyen imágenes como el logo del juego o la cara del personaje.

5.3.2. Música

Para la música quería un tipo de música alegre y rápida con toques rock-pop.

Encontré en *Jamendo* los grupos "No hair on head" y "Atomu". El primero es un autor español cuya canción "Jump!!" [7] me pareció acertada para el tema del menú principal. El segundo es un grupo japonés que también era del estilo que yo quería y utilicé su álbum "Vintage" [1] para los niveles y los créditos del juego.

Ambos grupos pueden ser escuchados y descargados libremente desde la página de *Jamendo*.

Ésta es la lista de temas:

- **Jump!!**: Tema introductorio del juego.
- **Atomic Power**: Tema para una fase.
- **Can I do**: Tema para una fase.
- **Makes it to memories**: Tema para una fase.
- **Music Life**: No usada. Forma parte del álbum de Atomu.
- **Mysterious Lip**: Tema para una fase.
- **SALA**: Tema para una fase.
- **When you were smiling**: Tema de los créditos.

5.3.3. Sonido

Para los efectos de sonido encontré la página *FreeSound.org*. Bajé algunos sonidos sueltos que posteriormente edité con el programa *Audacity*.

Las voces las grabó mi compañero Joaquín Jurado quien las modificó ligeramente con *Audacity*.

A continuación se listan los sonidos del juego:

- **Bloque**: Sonido que hace un bloque o cubo al ser destruido.
- **Crujido**: Crujido que suena al destruir un ermitaño.
- **Explosion_Aguda**: Explosión que suena al destruir una bola o globo pequeño.
- **Explosion_Grave**: Explosión que suena al destruir una bola o globo grande.

- **Explosion_Media**: Explosión que suena al destruir una bola o globo mediano.
- **Golpe**: Sonido del golpe recibido por el personaje cuando es tocado.
- **Graznido**: Graznido que emite un carpintero al ser destruido.
- **Navegacion**: Sonido que suena al subir o bajar entre las opciones del menú principal.
- **Pausa**: Sonido que suena al pausarse el juego.
- **Punto**: Sonido que suena al sumarse puntuación cuando se finaliza un nivel.
- **Seleccion**: Sonido que suena al seleccionarse una opción del menú.
- **Victoria**: Sonido no utilizado.

Y las voces:

- **Balloon_Breakers**: Voz que anuncia el título del juego al ser mostrado el menú principal.
- **Game_Over**: Voz que anuncia el fín de la partida.
- **Hurry_Up**: Voz que alerta sobre la necesidad de darse prisa porque queda poco tiempo para finalizar un nivel.
- **Ready_Go**: Voz que introduce un nivel.
- **Time_Over**: Voy que anuncia que el tiempo del nivel se ha agotado.

Capítulo 6

Implementación

A la hora de implementar el juego tuve claro desde un principio que utilizaría *C++* pues es un lenguaje muy potente y con muy buenos resultados en el mundo de los videojuegos. Además tenía la ventaja de conocer dicho lenguaje de la universidad.

Cómo ya he explicado anteriormente implementé todo el apartado gráfico con *Ogre 3D*. Existen numerosas formas de utilizar un motor gráfico y el caso de *Ogre 3D* no es diferente. Me basé en ejemplos que encontré en la web oficial de *Ogre 3D* para crear el primer incremento del juego hasta obtener una primera aplicación que arrancara el motor y permitiera una interacción básica con el juego.

Para el resto de elementos como la música y la lectura de ficheros XML utilicé *SDL Mixer* y *TinyXML*.

Una vez programada la lógica del juego el mayor reto del proyecto fue implementar el sistema de colisiones que explico en el siguiente apartado.

6.1. Colisiones

Las colisiones son un elemento fundamental en muchos videojuegos. En *Balloon Breakers* diseñé mis propios métodos para detectar colisiones ya que la física no es completamente real. Se producen y resuelven colisiones entre bolas, entre bolas y cubos, entre el personaje y los animales, y entre el personaje principal y las bolas.

6.1.1. Colisiones Bola-Bola

Para implementar el método ColisionesBolaBola de la clase Fase escribí un bucle anidado dentro de otro. Tanto el primer bucle como el segundo recorren el vector de bolas comparando su posición si no se trata de la misma bola y actuando en consecuencia si se produce una colisión.

Una colisión entre dos bolas es fácilmente detectable si realizamos el módulo del vector que va del centro de una a otra y restamos el valor por la suma de los radios de las dos bolas.

A continuación se expone el pseudocódigo de dicho método.

Sea **n** el número de elementos en el vector de punteros a objetos del tipo Bola:

```
desde i<-1 hasta n
  desde j<-1 hasta n
    si i<>j ^ (distancia(Bolas[i],Bolas[j])<(Bolas[i].radio()
    -Bolas[j].radio())) entonces
      alejar_bolas(Bolas[i],Bolas[j])
    fin_si
  fin_desde
fin_desde
```

6.1.2. Colisiones Bola-Cubo

La implementación del método ColisionesBolaCubo de la clase Fase es parecido a la del método ColisionesBolaBola. Sin embargo, mientras aquí el primer bucle recorre el vector de bolas, el segundo recorre el vector de cubos.

Para detectar la colisión entre una bola y un cubo tuve que utilizar la técnica divide y vencerás y dividir las posibles colisiones en varios casos diferentes dependiendo del lugar del cubo donde se produzca la colisión:

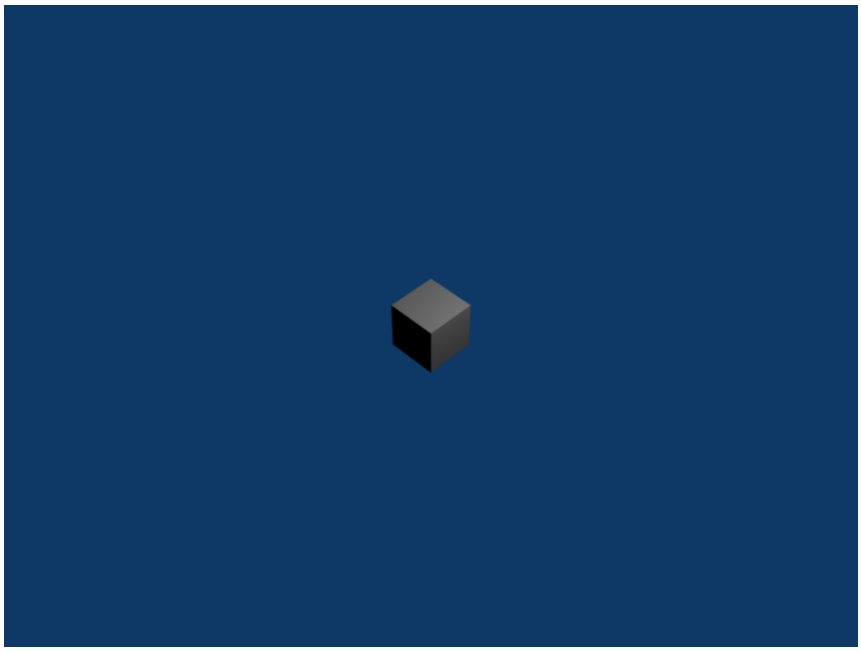

Figura 6.1: Cubo donde se produciría la colisión

- **Caso A: Colisiones con caras**
  Ocurre cuando el centro de la bola está directamente arriba, abajo, a la izquierda, a la derecha, detrás o delante del cubo y la distancia entre el centro del cubo y el centro de la bola es menor que el radio de la bola más 10 unidades (la mitad de la longitud de un lado).

Figura 6.2: Representación del área definida en el caso A

- **Caso B: Colisiones con vértices**
  Ocurre cuando el centro de la bola está sobre el cubo y a su derecha, izquierda, detrás o delante o cuando la bola está bajo el cubo y a su derecha, izquierda, detrás o delante. La colisión se produce cuando la distancia entre el centro de la bola y el punto más cercano de la arista es menor que el radio de la bola.

Figura 6.3: Representación del área definida en el caso B

- **Caso C: Colisiones con aristas**

Ocurre cuando la bola se encuentra en cualquier otra posición no descrita en los casos anteriores y la distancia entre el vértice más cercano y el centro de la bola sea menor que cinco.

Figura 6.4: Representación del área definida en el caso C

A continuación se expone el pseudocódigo de dicho método.

Sea **n** el número de elementos en el vector de punteros a objetos del tipo Bola y **m** el número de elementos en el vector de punteros a objetos del tipo Cubo:

```
desde i<-1 hasta n
  desde j<-1 hasta m
    //Caso A: colisiones con caras
    si Bolas[i].posicion_x<= Cubos[j].posicion_x+10
    ^  Bolas[i].posicion_x>= Cubos[j].posicion_x-10
    ^  Bolas[i].posicion_y<= Cubos[j].posicion_y+10
    ^  Bolas[i].posicion_y>= Cubos[j].posicion_y-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_z<= Cubos[j].posicion_z+10
    ^  Bolas[i].posicion_z>= Cubos[j].posicion_z-10
    ^  Bolas[i].posicion_y<= Cubos[j].posicion_y+10
    ^  Bolas[i].posicion_y>= Cubos[j].posicion_y-10
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_x<= Cubos[j].posicion_x+10
    ^  Bolas[i].posicion_x>= Cubos[j].posicion_x-10
    ^  Bolas[i].posicion_z<= Cubos[j].posicion_z+10
```

```
    ^  Bolas[i].posicion_z>= Cubos[j].posicion_z-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
  fin_desde
  desde j<-1 hasta m
    //Caso B: colisiones con vertices
    si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    sino_si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
```

```
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
  fin_desde
  desde j<-1 hasta m
    //Caso C: colisiones con aristas
    //Subcaso 1: Arriba, derecha
    si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 2: Arriba, izquierda
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x-10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 3: Arriba, norte
    sino_si Bolas[i].posicion_z < Cubos[j].posicion_z-10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_x < Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_x > Cubos[j].posicion_x-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 4: Arriba, sur
    sino_si Bolas[i].posicion_z > Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_x < Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_x > Cubos[j].posicion_x-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 5: Abajo, derecha
    sino_si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y-10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
```

```
    //Subcaso 6: Abajo, izquierda
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x-10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y-10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 7: Abajo, norte
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x-10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y-10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 8: Abajo, sur
    sino_si Bolas[i].posicion_z > Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y-10
    ^ Bolas[i].posicion_x < Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_x > Cubos[j].posicion_x-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 9: Centro, derecha, norte
    sino_si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z-10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 10: centro, izquierda, norte
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x-10
    ^ Bolas[i].posicion_z < Cubos[j].posicion_z-10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 11: Centro, derecha, sur
    sino_si Bolas[i].posicion_x > Cubos[j].posicion_x+10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
    //Subcaso 12: Centro, izquierda, sur
```

```
    sino_si Bolas[i].posicion_x < Cubos[j].posicion_x-10
    ^ Bolas[i].posicion_z > Cubos[j].posicion_z+10
    ^ Bolas[i].posicion_y < Cubos[j].posicion_y+10
    ^ Bolas[i].posicion_y > Cubos[j].posicion_y-10
      si distancia(Bolas[i],Cubos[j])<0
alejar_bola(Bolas[i])
salir_funcion
  fin_desde
fin_desde
```

6.1.3. Colisión Personaje-Bola

La implementación del método ColisionPersonajeBola de la clase Fase se basa en recorrer el vector de bolas y comprobar cada una de las bolas con dos esferas invisibles que se sitúan en el centro de la cintura del personaje y en su cuello. Esto se hace para agilizar las comparaciones ya que de otro modo habría que comprobar las colisiones con cada polígono que forma parte del personaje y esto sería muy costoso. A continuación se expone el código en C++ de dicho método:

```
bool Fase::ColisionPersonajeBola(void){
    bool colision=false;
    for (std::vector<Bola *>::iterator i = bolas.begin();
    (i!=bolas.end()&&!colision); ++i) {
        if ((((Vector3((*i)->sceneNode->getPosition()-
(personaje->sceneNode->getPosition()+
Vector3(0,5,0)))).length()-(*i)->getRadio())<5)||
        (((Vector3((*i)->sceneNode->getPosition()-
(personaje->sceneNode->getPosition()+
Vector3(0,20,0)))).length()-(*i)->getRadio())<5))
            colision=true;
    }
    return colision;
}
```

6.1.4. Colisión Personaje-Carpintero y Personaje-Ermitaño

La implementación de estos métodos se basan en recorrer los vectores vector<Carpintero *>o vector <Ermitano *>y comprobar las colisiones entre una esfera invisible que rodea a cada ermitaño o pájaro carpintero con las dos esferas invisibles del personaje mencionadas anteriormente.

A continuación se expone el código en C++ de dicho método:

```
bool Fase::ColisionPersonajeCarpintero(void){
    bool colision=false;
    for (std::vector<Carpintero *>::iterator i =
    carpinteros.begin(); (i!=carpinteros.end()&&!colision);
    ++i) {
        if ((((Vector3((*i)->sceneNode->getPosition()-
(personaje->sceneNode->getPosition()+
```

```
Vector3(0,5,0)))).length())<10)||
        (((Vector3((*i)->sceneNode->getPosition()-
(personaje->sceneNode->getPosition()+
Vector3(0,20,0)))).length())<10))
            colision=true;
    }
    return colision;
}

bool Fase::ColisionPersonajeErmitano(void){
    bool colision=false;
    for (std::vector<Ermitano *>::iterator i =
    ermitanos.begin(); (i!=ermitanos.end()&&!colision);
    ++i) {
        if ((((Vector3((*i)->sceneNode->getPosition()-
(personaje->sceneNode->getPosition()+
Vector3(0,5,0)))).length())<10)||
        (((Vector3((*i)->sceneNode->getPosition()-
(personaje->sceneNode->getPosition()+
Vector3(0,20,0)))).length())<10))
            colision=true;
    }
    return colision;
}
```

Capítulo 7

Pruebas

Al haber seguido un desarrollo incremental he ido probando durante cada incremento las nuevas funcionalidades añadidas.

A continuación explico las pruebas que he realizado para cada incremento.

7.1. Primer incremento

- Se dibuja un escenario simple con el personaje.

  Tuve problemas con el tamaño de los elementos que resolví aplicando todos los cambios en la deformación de cada objeto en *Blender* antes de exportar.

- Es posible mover al personaje.

  El movimiento no me dió muchos problemas. Modifiqué el código para que el personaje se desplazara a la velocidad adecuada.

- El personaje no sale de la pantalla.

  Añadí una condición extra para que cuando el personaje llegue a una posición determinada no pueda seguir avanzando en esa dirección.

7.2. Segundo incremento

- Se carga el menú principal.

  No tuve ningún problema aquí.

- Se carga la pantalla de créditos.

No tuve ningún problema aquí.

- Se carga el escenario fase.

  No tuve ningún problema aquí.

- Se cambia de una escena a otra.

  No tuve ningún problema aquí.

- Los globos o bolas colisionan unos con otros.

  Hubo un pequeño problema y es que las bolas entraban una dentro de la otra cuando se producia una colisión. Esto lo resolví haciendo que las bolas se alejen entre sí dependiendo de sus posiciones relativas y no del lugar donde se produce la colisión.

- Los globos o bolas colisionan con el personaje y se pierde una vida cuando esto ocurre.

  No tuve ningún problema aquí.

- Se muestra el mensaje Game Over cuando no quedan más vidas.

  No tuve ningún problema aquí.

- Es posible disparar el arpón y este aparece en la posición correcta.

  En principio el arpón aparecía justo en medio del personaje. Cambié esto para que apareciera justo en medio del arma del personaje.

- Aparecen los elementos de la capa superior como son el número de vidas, tiempo, nivel...

  Tuve que modificar el script de la capa superior para que los elementos aparecieran en un lugar adecuado y que sus colores no se confundieran con los fondos del juego.

- El tiempo disminuye adecuadamente.

  No tuve ningún problema aquí.

- El arpón destruye las bolas.

  No tuve ningún problema aquí.

- La puntuación aumenta adecuadamente cuando se destruye una bola.

  No tuve ningún problema aquí.

7.3. Pruebas finales

- Se cargan las fases con su escenario y luz.

  No tuve ningún problema aquí.

- Se cargan las bolas, cubos y animales en la posición indicada en el archivo XML.

  Tuve que modificar el centro de los animales en *Blender* para que los ermitaños aparecieran en una posición adecuada.

- Los globos o bolas colisionan con los cubos.

  Este fué el punto más desafiante para mí. Cómo ya comenté en capítulos anteriores deseché la idea de usar *Bullet* para la física del juego y la implementé yo mismo. La implementación de la física se complicó mucho cuando incluí los cubos en el juego y tuve que diseñar la función de colisiones bolas-cubos. Finalmente el resultado fué satisfactorio.

- Los globos o bolas colisionan con los animales.

  No tuve ningún problema aquí.

- El personaje colisiona con los animales.

  No tuve ningún problema aquí.

- La puntuación aumenta adecuadamente cuando se destruye un animal.

  No tuve ningún problema aquí.

- Se muestran los efectos de partícula adecuados dependiendo de su emisor y la posición de éste.

  No tuve ningún problema aquí.

- Se escuchan los sonidos y música del juego correctamente.

  En principio no podía escuchar nada. Convertí los ficheros de música a formato .ogg con *Audacity* y *SDL mixer* los reprodució sin problemas.

Capítulo 8

Conclusiones

8.1. Desarrollo del proyecto

Durante el proyecto he conseguido básicamente mi principal objetivo: obtener los conocimientos básicos para crear un videojuego y saber dónde y cómo encontrar la información necesaria para ello.

He aprendido cómo utilizar diversas aplicaciones de código abierto que también pueden ser necesarias en otro tipo de proyectos. Las diferentes tecnologías utilizadas me han parecido en general muy buenas y creo que se pueden obtener resultados muy buenos con ellas. A continuación describo mi opinión de estas tecnologías:

- **C++:** conocía este lenguaje de la carrera y sabía de su uso en el mundo profesional de desarrollo de videojuegos. C++ me ha parecido muy potente y fácil de codificar con él.
- **Ogre 3D:** tiene una curva de aprendizaje un poco compleja pero siguiendo los tutoriales y gracias a su amplia comunidad no es difícil hacerse con su manejo. Una posible crítica de mi parte es que la mayoría de los tutoriales, ejemplos y aplicaciones están orientados a ser compilados con Visual Studio, herramienta privada de pago de Microsoft.
- **OIS:** aunque no he llegado a utilizar OIS en profundidad esta biblioteca me ha ofrecido lo que necesitaba para controlar el teclado.
- **SDL y SDL mixer:** he utilizado SDL mixer para el sonido del juego. SDL mixer me ha ofrecido todo lo necesario para la gestión de música, voces y efectos sonoros. Me ha parecido muy fácil de utilizar aunque aún no sea completamente orientado a objetos.
- **TinyXML:** me ha permitido hacer exactamente lo que quería, leer archivos XML. La biblioteca es bastante más compleja y permite también escritura en documentos XML, entre otras cosas.
- **Blender:** conocía 3D-Studio Max desde hace tiempo y comencé a utilizar Blender hace 2 años. Aunque al principio parece muy complejo dado su interfaz de usuario tan diferente con el tiempo es posible trabajar con él de una forma rápida y con buenos resultados.
- **GIMP:** me ha parecido un programa con muchas posibilidades. La mayor crítica que pueda hacerle es tu compleja interfaz de usuario y la no existencia de un modo con una única ventana aunque al parecer este último se arreglará en la próxima versión.
- **Audacity:** Audacity me ha permitido de una forma fácil modificar los archivos de música y sonido para el videojuego. Aunque no he trabajado muy a fondo con él me ha parecido una herramienta muy intuitiva y con muchas opciones.

- **DIA:** sencillo y práctico de utilizar. Aunque aún no se encuentre en su versión 1.0 DIA tiene todo lo necesario para realizar muchos tipos de documentos técnicos, entre ellos todos los diagramas UML que he utilizado en el proyecto. Una pequeña pega es que la configuración de usuario no se guarda y es necesario reconfigurar algunos aspectos cada vez que se abre la aplicación.
- **Planner:** con Planner he realizado los diagramas de Gantt de esta documentación. Es una herramienta sencilla y fácil de utilizar. Contiene además muchas características para gestionar recursos que no he utilizado al ser este un proyecto para una sola persona.
- **LaTeX:** el resultado de LaTeX no tiene comparación con el resultado de un procesador de textos convencional. Con LaTeX he conseguido un acabado para esta memoria.

Finalmente, he quedado satisfecho con el resultado final del juego. Participé con él en el V concurso Universitario de software libre obteniendo la mención "Finalista mejor proyecto de ocio" [1] aunque me gustaría contar con más tiempo para futuras ampliaciones que comento a continuación.

8.2. Posibles ampliaciones

Aunque se ha acabado una primera versión del videojuego, como en muchos proyectos, es posible ampliarlo y mejorarlo. A continuación, se incluye una lista con posibles ampliaciones y mejoras:

- **Modo para dos o más jugadores:** sería posible ampliar el juego ofreciendo la posibilidad de jugar con más de un jugador simultáneamente. El diseño del resto de personajes podría estar basado en el personaje principal y cambiar simplemente la textura o algunas características básicas del modelo como el arma, pañuelo, etcétera. Habría que implementar la gestión de la entrada para que se pudiera elegir teclas, mandos u otro tipo de dispositivo y modificar la clase Fase y OyenteFase para que controlara a más de un jugador. Además habría que modificar ligeramente el Overlay de las fases para que se pudiera mostrar la información relevante de cada jugador (su puntuación y vidas).
- **Nuevas fases:** sería posible añadir nuevos niveles con escenarios diferentes. Esta sería una de las ampliaciones más sencillas puesto que sólo sería necesario crear el escenario 3D, buscar la música adecuada y crear los correspondientes archivos XML que contienen la descripción del nivel.
- **Guardar partida:** también se podría implementar una opción para guardar una partida y continuarla. Se podría dar la opción de guardar una partida justo después de avanzar un nivel para así no tener que guardar las posiciones, velocidad, etcétera de los elementos de la fase. Se podría añadir la opción continuar partida en el menú principal que abriría un nuevo menú con las diferentes partidas guardadas.
- **Otras mejoras:** de relativa menor importancia pero también a tener en cuenta serían otras mejoras como escaleras por donde el personaje pudiera subir y bajar a las plataformas, armas diferentes que el personaje pueda ir recogiendo u otros objetos como frutas, vidas extra, etcétera.

[1] http://www.uca.es/es/cargarAplicacionNoticia.do?identificador=2826

Apéndices

Apéndice A

Instalación

A.1. Instalación en GNU/Linux

En esta sección se proporcionan las indicaciones pertinentes para que puedas descargar e instalar Balloon Breakers en tu sistema. En primer lugar, debes acudir a la sección de ficheros del repositorio del juego en la forja de RedIRIS:

https://forja.rediris.es/scm/?group_id=755

En segundo lugar hay que obtener el compilador GCC y la herramienta make introduciendo la siguiente orden en la terminal:

```
sudo apt-get install gcc g++ make
```

El siguiente paso es instalar *Ogre*. Para ello añadimos el repositorio introduciendo el comando:

```
sudo add-apt-repository ppa:ogre-team/ogre
```

Posteriormente actualizamos el sistema:

```
sudo apt-get update
```

A continuación instalaremos *OIS*, sistema del cual depende *Ogre 3D* introduciendo el siguiente comando:

```
sudo apt-get install libois-dev
```

El siguiente paso es obtener *SDL* y *SDL Mixer*. Para ello introducimos el siguiente comando en la terminal:

```
sudo apt-get install libsdl1.2-dev libsdl-mixer1.2-dev
```

Por último, para obtener el código y todos los archivos necesarios para compilar, enlazar y ejecutar *Balloon Breakers* es necesario obtener Subversion:

```
sudo apt-get install subversion
```

Una vez instalado subversion procedemos a la descarga del proyecto introduciendo la siguiente orden:

```
svn checkout https://forja.rediris.es/svn/balloonbreakers
```

A.2. Instalación en Windows

Existe una versión precompilada para Windows que se puede descargar desde:

http://forja.rediris.es/frs/?group_id=755&release_id=1671

El juego se encuentra dentro de un archivo .rar que es necesario descomprimir para que se ejecute correctamente.

Una vez descargado el juego basta con hacer doble clic sobre el archivo .exe que es encuentra en la carpeta raiz de Balloon Breakers

Apéndice B

Userguide

Userguide

Thanks for obtaining Balloon Breakers

Introduction

Balloon Breakers is a platforms game where you have to shoot and destroy all balloons that appear in every of its 15 levels.

In every level there are also animals as crabs and woodpeckers that will destroy the balls or our hero if they touch them. You can shoot them to get extra points.

Destroy the platforms with the harpoon to make the levels easier and get extra points!

Controls

To control the character and move through the main menu you only need a keyboard.

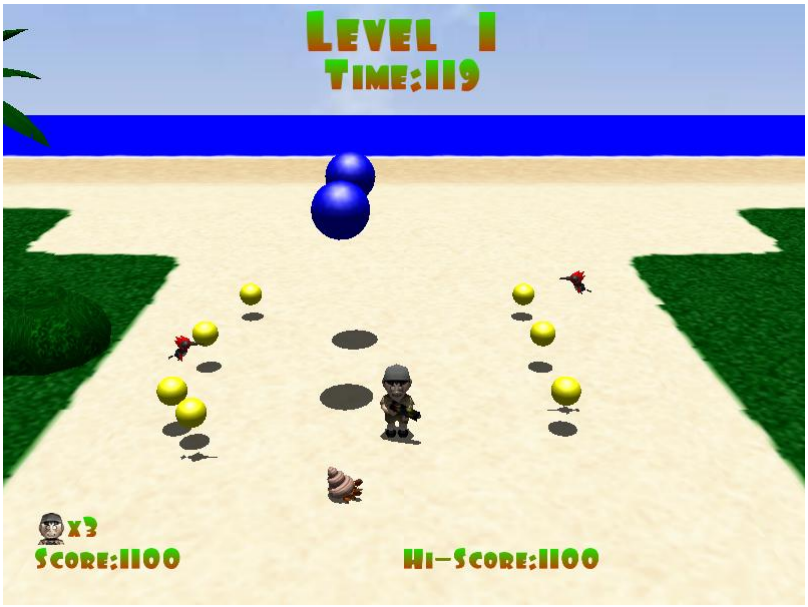

Figura B.1: Playing Balloon Breakers

In the main menu push **UP** or **DOWN** to navigate through the options. Press ENTER to select the highlighted option.

Figura B.2: Main menu

During the gameplay of Balloon Breakers you can move the character pressing UP, DOWN, LEFT or RIGHT.

To shoot the harpoon press **SPACE**.

Figura B.3: Shooting

You can also change the view by pressing **F1**, **F2**, **F3** or **F4**.

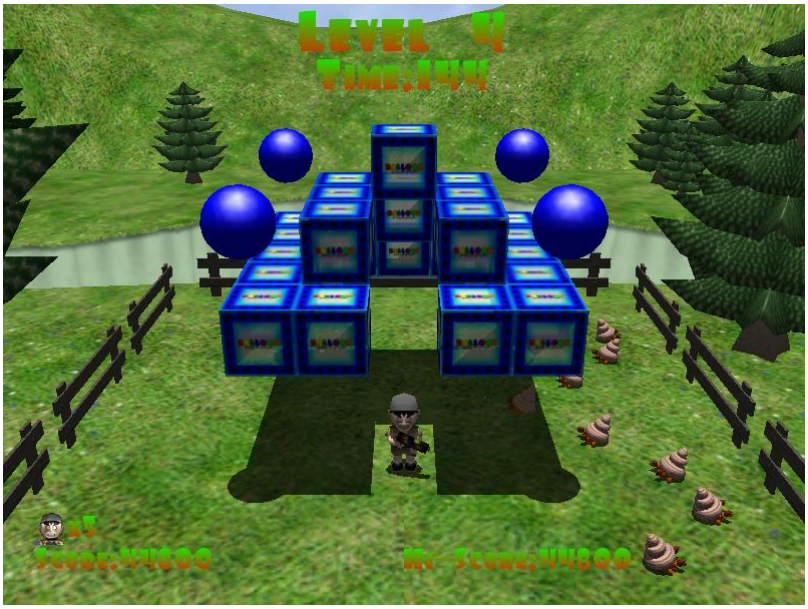

Figura B.4: First view

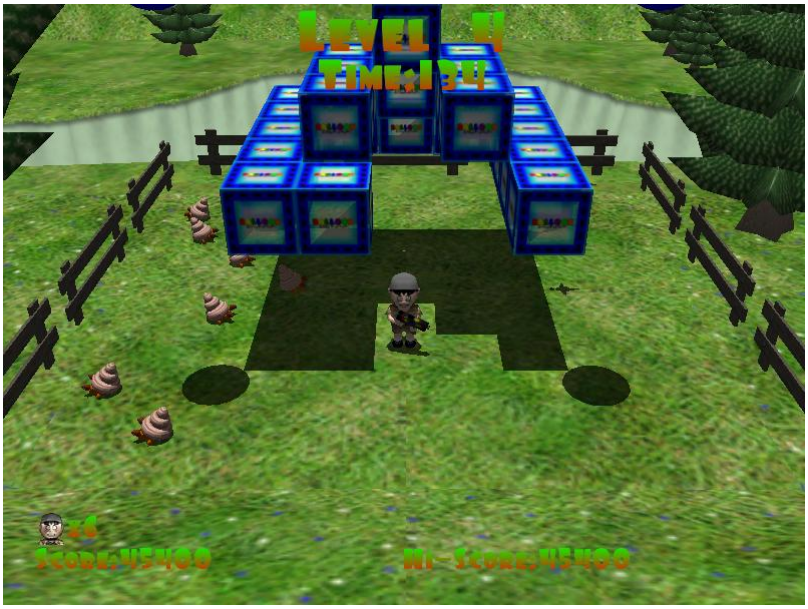

Figura B.5: Second view

Figura B.6: Third view

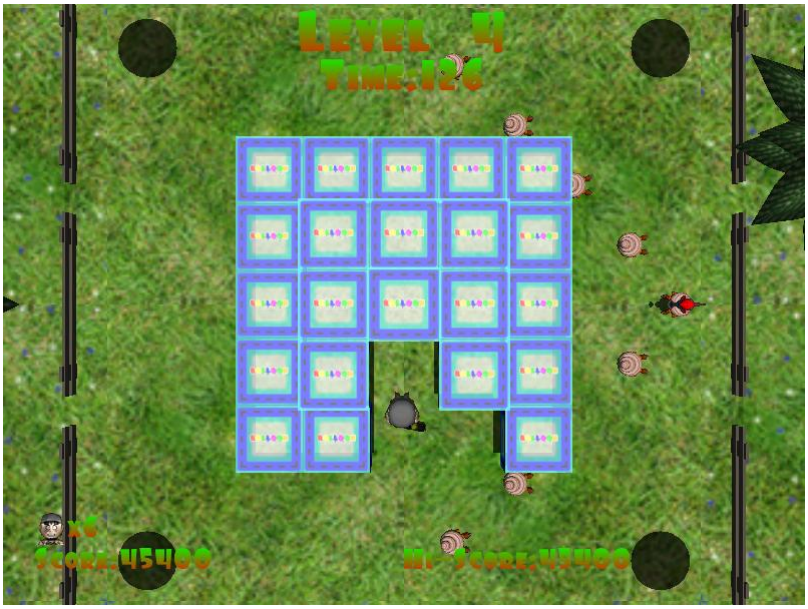

Figura B.7: Fourth view

To pause the game press **ENTER**.

Figura B.8: Paused game

If you want to return to the main menu press **ESC**.

I wish you a lot of fun playing Balloon Breakers

Apéndice C

Guía de usuario

Guía de usuario

Gracias por obtener Balloon Breakers.

Introduction

Balloon Breakers es un videojuego de plataformas donde tienes que disparar y destruir todos los globos que aparecen en cada uno de sus 15 niveles.

En cada nivel hay también animales como cangrejos ermitaños y pájaros carpinteros que destruirán los globos o a nuestro héroe, al tocarlos. Puedes dispararlos para conseguir puntuación extra.

Destruye las plataformas con el arpón para hacer los niveles más fáciles y conseguir puntos extra!

Controls

Para controlar al personaje y moverse por el menú principal sólo se necesita un teclado.

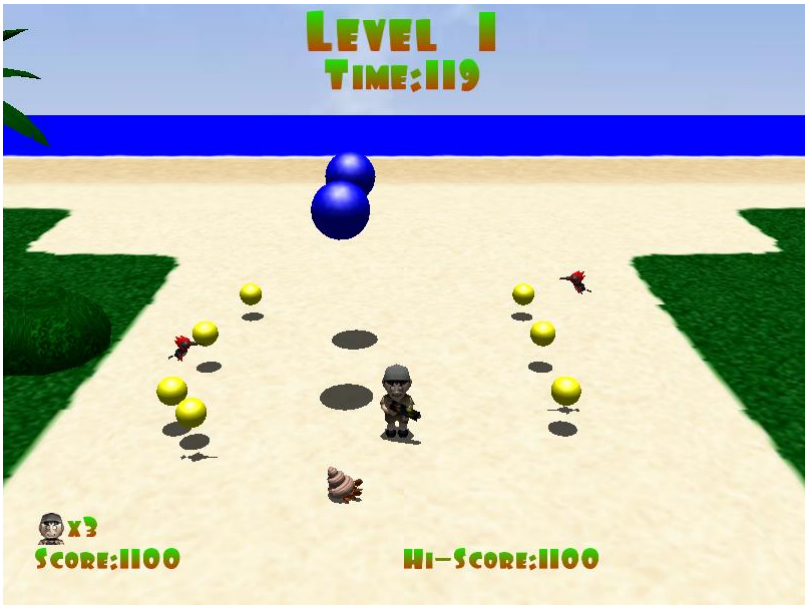

Figura C.1: Jugando Balloon Breakers

En el menú principal presiona **ARRIBA** o **ABAJO** para desplazarte por las opciones. Pulsa ENTER para seleccionar la opción resaltada.

Figura C.2: Menú principal

Durante el juego se puede mover el personaje presionando **ARRIBA**, **ABAJO**, **IZQUIERDA** o **DERECHA**.

Pulsa **ESPACIO** para disparar el arpón.

Figura C.3: Disparando

También puedes cambiar la vista pulsando la tecla **F1**, **F2**, **F3** o **F4**.

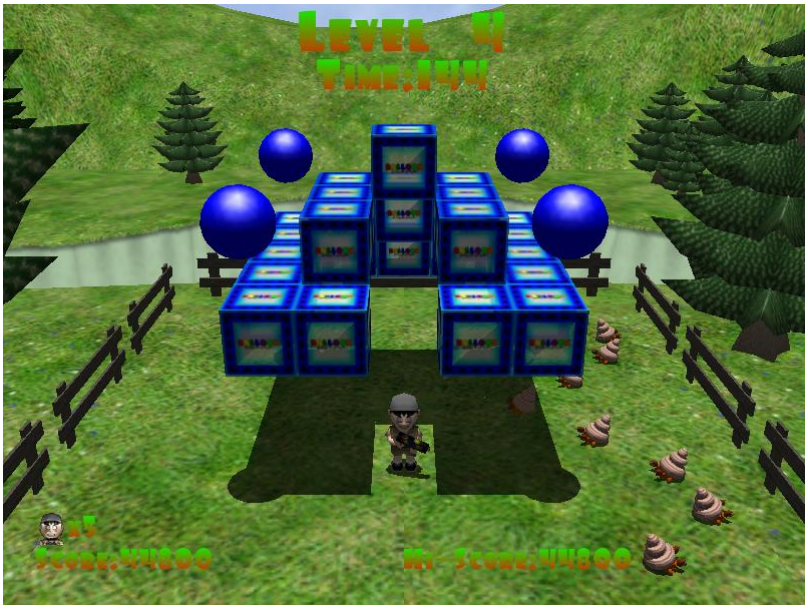

Figura C.4: Primera vista

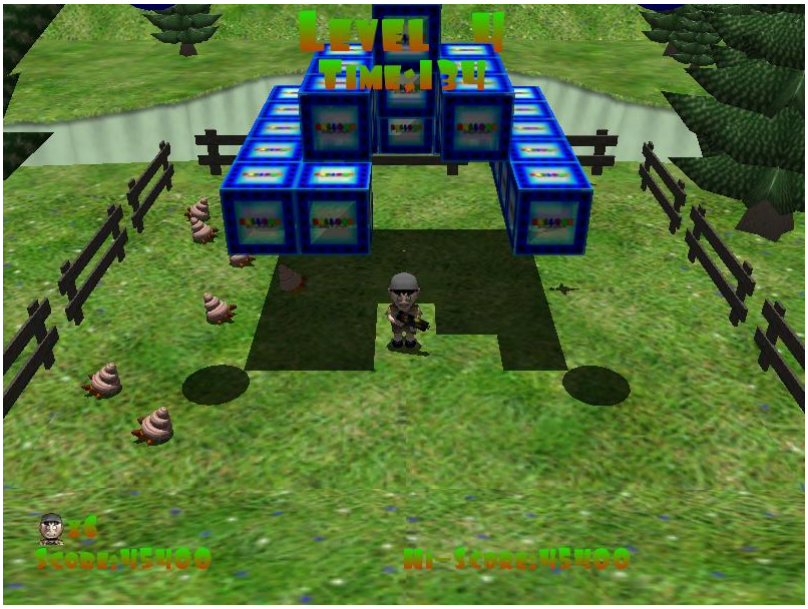

Figura C.5: Segunda vista

Figura C.6: Tercera vista

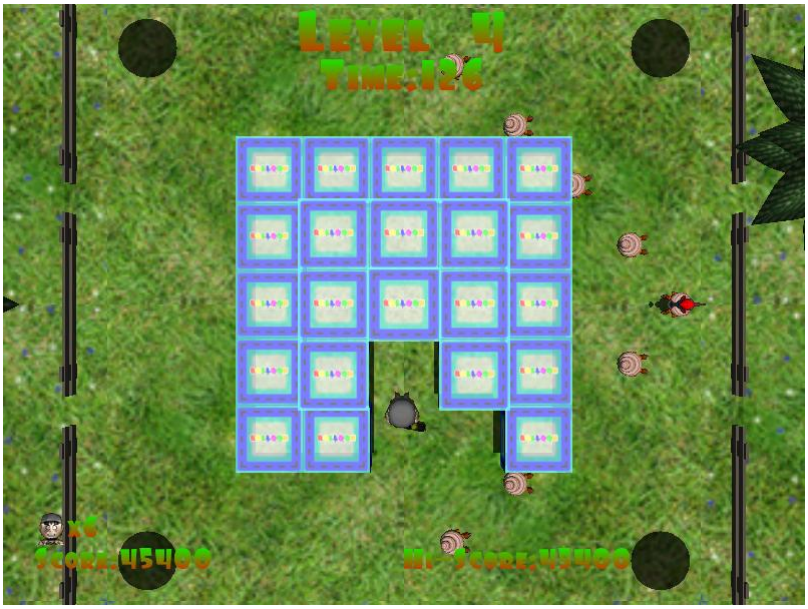

Figura C.7: Cuarta vista

Para poner en pausa el juego presiona **ENTER**.

Figura C.8: Juego pausado

Si deseas volver al menú principal presiona la tecla **ESCAPE**.

Espero que lo pases bien jugando a Balloon Breakers

Bibliografía

[1] Atomu. Atomu - Vintage en Jamendo.com. http://www.jamendo.com/es/album/63940.

[2] Diversos autores. Wikipedia. http://es.wikipedia.org/videojuego/.

[3] Varios autores. Freesound. http://www.freesound.org/.

[4] Varios autores. Wikipedia. http://es.wikipedia.org/wiki/Software#Modelo_iterativo_incremental.

[5] Mercè Galán. *Blender. Curso de Iniciación*. Inforbook's, S.L., 2007.

[6] Francisco Palomo Gerardo Aburruzaga, Inmaculada medina. *Fundamentos de C++*. Servicio de Publicaciones de la Universidad de Cádiz, 2009.

[7] No hair on head. No hair on Head - Jump! en Jamendo.com. http://www.jamendo.com/es/album/53402.

[8] Gregory Junker. *Pro OGRE 3D Programming*. Apress, 2006.

[9] Usuario: koffiepad. Moby Games. http://www.mobygames.com/game/msx/cannon-ball/screenshots/gameShotId,131577/.

[10] Olof Kylander. *Gimp: The Official Handbook*. Coriolis Group Books, 1999.

[11] Ernest Pazera. *Focus On SDL*. Course Technology PTR, 2002.

[12] Carla Schroder. *The Book of Audacity*. No Starch Press, 2011.

[13] The Ogre Team. Ogre 3D. http://www.ogre3D.com/.

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
The "**publisher**" means any person or entity that distributes copies of the Document to the public.
A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.
You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.
However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.
"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.
An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.
The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.